Reinforcement Learning: Theory, methods and application to decision support systems

by

Hildegarde Suzanne Mouton

Thesis presented
in partial fulfilment of the requirements for the degree of
Master of Science in Applied Mathematics
at the
University of Stellenbosch

Department of Applied Mathematics,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Supervisor: Prof B.M. Herbst
Co-Supervisor: Dr. J.H.S. Roodt

December 2010

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: .....
H.S Mouton

Date: .....

Copyright © 2010 University of Stellenbosch
All rights reserved.

Abstract

In this dissertation we study the machine learning subfield of Reinforcement Learning (RL). After developing a coherent background, we apply a Monte Carlo (MC) control algorithm with exploring starts (MCES), as well as an off-policy Temporal-Difference (TD) learning control algorithm, Q-learning, to a simplified version of the Weapon Assignment (WA) problem.

For the MCES control algorithm, a discount parameter of $\gamma = 1$ is used. This gives very promising results when applied to 7×7 grids, as well as 71×71 grids. The same discount parameter cannot be applied to the Q-learning algorithm, as it causes the Q-values to diverge. We take a greedy approach, setting $\epsilon = 0$, and vary the learning rate (α) and the discount parameter (γ). Experimentation shows that the best results are found with α set to 0.1 and γ constrained in the region $0.4 \leq \gamma \leq 0.7$.

The MC control algorithm with exploring starts gives promising results when applied to the WA problem. It performs significantly better than the off-policy TD algorithm, Q-learning, even though it is almost twice as slow.

The modern battlefield is a fast paced, information rich environment, where discovery of intent, situation awareness and the rapid evolution of concepts of operation and doctrine are critical success factors. Combining the techniques investigated and tested in this work with other techniques in Artificial Intelligence (AI) and modern computational techniques may hold the key to solving some of the problems we now face in warfare.

Opsomming

Die fokus van hierdie verhandeling is die masjienleer-algoritmes in die veld van versterkingsleer. 'n Koherente agtergrond van die veld word gevolg deur die toepassing van 'n Monte Carlo (MC) beheer-algoritme met ondersoekende begintoestande, sowel as 'n afbeleid Temporale-Verskil beheer-algoritme, Q-leer, op 'n vereenvoudigde weergawe van die wapentoekenningsprobleem.

Vir die MC beheer-algoritme word 'n afslagparameter van $\gamma = 1$ gebruik. Dit lewer belowende resultate wanneer toegepas op 7×7 roosters, asook op 71×71 roosters. Dieselfde afslagparameter kan nie op die Q-leer algoritme toegepas word nie, aangesien dit veroorsaak dat die Q-waardes divergeer. Ons neem 'n gulsige aanslag deur die gulsigheidsparameter te verstel na $\epsilon = 0$. Ons varieer dan die leertempo (α) en die afslagparameter (γ). Die beste eksperimentele resultate is behaal wanneer $\alpha = 0.1$ en as die afslagparameter vasgehou word in die gebied $0.4 \leq \gamma \leq 0.7$.

Die MC beheer-algoritme lewer belowende resultate wanneer toegepas op die wapentoekenningsprobleem. Dit lewer beduidend beter resultate as die Q-leer algoritme, al neem dit omtrent twee keer so lank om uit te voer.

Die moderne slagveld is 'n omgewing ryk aan inligting, waar dit kritiek belangrik is om vinnig die vyand se planne te verstaan, om bedag te wees op die omgewing en die konteks van gebeure, en waar die snelle ontwikkeling van die konsepte van operasie en doktrine lei tot sukses. Die tegniekes wat in die verhandeling ondersoek en getoets is, en ander kunsmatige intelligensie tegnieke en moderne berekeningstegnieke saamgesnoer, mag dalk die sleutel hou tot die oplossing van die probleme wat ons tans in die gesig staar in oorlogvoering.

Acknowledgement

I would like to thank the following people:

- Prof B.M. Herbst for his guidance as supervisor.
- Dr J.H.S. Roodt at the CSIR for his guidance as co-supervisor and mentor as well as providing the funding for this project.
- My fellow colleagues for being so supportive and available to exchange problems and ideas.
- All the lecturers along my path from whom I have learnt the basics needed in order to complete this dissertation.
- Devan Koen for his love and support.
- My parents for believing in me and teaching me that nothing in this world is impossible. This one's for you, Dad.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

Symbols

Ambiguous symbols will become clear within the context they are used.

t	= discrete time step
T	= final time step of an episode
s_t	= state at time t
a_t	= action at time t
r_{t+1}	= reward received when in state s at time t
R_t	= expected return following t
$R_t^{(n)}$	= n-step return following t
R_t^{λ}	= λ-return following t
π	= policy
$\pi(s)$	= action taken in state s under deterministic policy π
$\pi(s,a)$	= probability of taking action a when in state s under stochastic policy π
$\mathcal{S}$	= set of all nonterminal states
$\mathcal{S}^+$	= set of all states, including terminal state
$\mathcal{A}(s)$	= set of actions possible in state s
$\Re$	= set of real numbers
$\mathcal{P}_{ss'}^a$	= probability of each possible next state, s', given a state s and action a
$\mathcal{R}_{ss'}^a$	= expected immediate reward, given a state s, action a and next state s'
$V^\pi(s)$	= state-value function for policy π
$V^*(s)$	= optimal state-value function
V, V_t	= estimates of V^π or V^*
$Q^\pi(s,a)$	= action-value function for policy π
$Q^*(s,a)$	= optimal action-value function

Q, Q_t = estimates of Q^π or Q^*
$e(s, a)$ = eligibility trace for state-action pair

γ = discount rate parameter
α = step-size parameter
ϵ = probability of random action in ϵ-greedy policy
λ = decay rate parameter for eligibility traces
δ_t = temporal-difference error at time t
X = grid size in weapon assignment problem

List of Abbreviations

AA	Anti-Aircraft
ACO	Ant Colony Optimization
AD	Air Defence
AI	Artificial Intelligence
AOR	Area of Responsibility
BN	Bayesian Network
C2	Command and Control
CPA	Closest Point of Approach
DA	Defended Asset
DP	Dynamic Programming
DSS	Decision Support System
E-O	Electro-optical
EW	Electronic Warfare
FCO	Fire Control Operator
GA	Genetic Algorithm
GBAD	Ground Based Air Defence
GBADS	GBAD system
GPI	Generalized Policy Iteration
HCI	Hostility Classification/Identification
HMI	Human Machine Interface
IPB	Intelligence Preparation of the Battlefield
IR	Infra-Red
IT	Information Technology
LAN	Local Area Networks
LOS	Line-of-Sight
MC	Monte Carlo
MCES	Monte Carlo with Exploring Starts
MDP	Markov Decision Processes
OODA	Observe-Orient-Decide-Act
OP	Observation Post(s)
OPFOR	Opposing Force

RF	Radio Frequency
RL	Reinforcement Learning
RST2	Random Search Technique
SA	Situation Awareness
SAM	Surface-to-Air Missile
Sarsa	State-Action-Reward-State-Action
TCI	Type Classification/Identification
TD	Temporal-Difference (learning)
TE	Threat Evaluation
TEWA	Threat Evaluation and Weapon Assignment
TM	Track Management
UAV	Unmanned Aerial Vehicle
VOI	Volume of Interest
VSHORAD	Very SHOrt Ranged Air Defence
WA	Weapon Assignment
WSEM	Weapon System Efficiency Matrix
WTA	Weapon Target Assignment

Chapter 1

Introduction

1.1 The Problem

From across the distance an airplane flies into sight of the weapons' master. The message comes in from the detection unit: it is an enemy plane. There are four manned missiles on the ground defending the hangar. The weapons' master needs to make a decision, almost without time to think: which of the four missiles must shoot at the enemy plane? And if that missile misses, which one shoots next? The wrong choice could be fatal: not only could this endanger the soldiers manning the missiles, but also the hangar, and it could lead to the elimination of the very asset he wishes to defend.

Furthermore, with every new mission the layout changes and a new approach may be needed. Typical military doctrine is devised to cover most scenarios by generalising the approach. In this dissertation we consider Reinforcement Learning (RL) as a method to discover specific doctrinal approaches at the level of Weapon Assignment (WA).

1.2 Review of approaches to the problem

From the available literature on the WA problem, the most popular approaches to solving the problem seem to be integer programming (Rosenberger *et al.* (2005), Deep and Plant (2005)), Genetic Algorithms (GAs) (Grant, 1993), heuristics (Madni and Andrecut, 2009) and hybrid methods consisting of GAs and heuristics (Lee and Lee, 2005). Other approaches found were Bayesian networks (BNs) (Le Roux *et al.*, 2006), Dynamic Programming (DP) (Sikanen, 2008), and only one case in which RL was used (Azak and Bayrak, 2008).

Rosenberger *et al.* (2005) extend the basic WA problem by allowing for multiple threat assignments per platform. They formulate the problem as a linear integer programming problem and investigate two solution methods. The first method is a greedy approach based

on the sequential application of the auction algorithm, generalised for assigning weapons to threats. Auction algorithms refer to several variations of a combinatorial optimisation algorithm which solves assignment problems (Bertsekas, 1979).

The second method is built on a branch-and-bound framework that enumerates feasible tours of resources. *Branch-and-bound* is a method that finds the optimal solution to an integer programming problem by efficiently enumerating the points in a subproblem's feasible region (Winston, 2004). They compare the two methods and investigate the suitability and performance of the successive auction algorithm adopted for a class of multi-threat assignment problems. They also extend the branch-and-bound technique to assigning multiple platforms per threat.

Rosenberger *et al.* (2005) perform several simulations to compare the sequential method with the branch-and-bound method. They find that the use of successive auctions beyond the main weapon-threat pairings to generate multi-threat assignments produce good results overall. The branch-and-bound method can be used to provide efficient solutions to the multi-weapon assignment problem which may be obtained quickly through the use of heuristic benefit threshold values and differences.

Deep and Plant (2005) model the WA problem as a non-linear integer programming problem. There are constraints on various types of weapons available and on the minimum number of weapons to be assigned to various threats. The objective function is non-linear, the constraints are linear, and the decision variables are restricted to be integers. This resulted in the I-GRST algorithm (algorithm for integer and mixed integer optimisation problems).

The I-GRST algorithm is based on the GRST algorithm, which in turn is the combination of the GA approach and the random search technique (RST2) of Mohan and Shanker (1994), which is a controlled random search technique for global optimisation using quadratic approximation. Based on results, it is concluded that as compared to the value of the objective function in the source, the value of the objective function obtained is generally better.

Sikanen (2008) approaches the WA problem with Dynamic Programming (DP). The example problems show the benefit of cooperation in risky environments. One- and two-step lookahead solutions are measured against the full DP solution and it is found that the two-step lookahead performs almost as well as the full DP for most situations, finding the solution in about half the time. One-step lookahead is dramatically faster than the full DP and produced only slightly worse solutions. Two-step lookahead performed almost as well as the full DP solution but was significantly faster.

Le Roux *et al.* (2006) use a classifier-based approach and model the WA problem with a Bayesian decision Network (BN). The assignment patterns of an existing WA subsystem can be used as training and test data for a classifier. The aim is to determine the probability distribution over the states of the hidden variable "Assignment" over time, given time series data of the children nodes (observed variables or features). The training data set in this case is generated by simulating a number of different system scenarios and recording the

WA.

The results are very promising for modelling WA with a BN. BNs are desirable for their probabilistic nature and their ability to learn from experience data, but BNs used as a classifier may not be the most suitable method for an optimisation problem as it is very difficult to model features of a continuous nature. The results show that the prioritisation of weapon-threat pairs can successfully be determined in a simple scenario.

Grant (1993) investigates the effectiveness of GAs for solving the WA problem. A GA is a search technique used to find exact or approximate solutions to optimisation and search problems (Goldberg, 1989). GAs are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology. The schedules are encrypted into strings, and multiple weapons are represented by substrings. The probability of defended asset survival against multiple threats is calculated with a multidimensional cost function. Given both a random and weighted initial population of 10 schedules, each with 12 decision instants that uniformly describe a 60 second time interval, a GA is applied to 10 scenarios, with one or two threats arriving to determine an optimal schedule. The GA determines the optimal schedule 81% of the time on average. The quality of schedules derived from random or weighted initial populations was nearly equal. In all cases, the GA quickly delivers valid and near optimal schedules for time-critical threat engagements.

Madni and Andrecut (2009) modelled the problem with two heuristic algorithms based on simulated annealing and threshold accepting methods. Their computational results show that by using these algorithms, relatively large instances of the WA problem can be solved near-optimally in a few seconds on a standard personal computer.

A hybrid search algorithm with heuristics for a resource allocation problem encountered in practice, such as WA, is proposed by Lee and Lee (2005). The proposed algorithm has both the advantages of GA and Ant Colony Optimisation (ACO) (Dorigo, 1992) that can explore the search space and exploit the best solution. ACO is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs.

From simulation results it is deduced that the proposed algorithm performs well for tested problems. Because of the complementary properties of GA and ACO, the hybrid approach outperforms other existing algorithms.

Azak and Bayrak (2008) implement learning agents for decision making in TEWA problems of C2 systems. The goal for the project is to optimise the performance in decision making for TEWA problems of multi-armed platforms. Focusing only on their approach to the WA problem, we see the use of the RL technique Q-learning to solve this WA problem. After training, the agents learn how to coordinate with other agents and how to select a threat to destroy in any state of the defence system.

They conclude that agents benefit from coordination and they can complete scenarios more successfully than independent agents. In addition, they see that their default Q-learning algorithm implementation for WA without coordination is also successful and agents be-

come more mature with sufficient number of training data. Although coordination seems to improve agents performance, it can be claimed that it is not a sufficient experimental achievement.

1.3 Reinforcement Learning

Our approach is to apply methods from RL to WA, as we are aware of only one RL approach in the current literature, which is (Azak and Bayrak, 2008). We develop the basic concepts of RL and illustrate them with simple examples and finally apply RL to a simple WA problem.

This dissertation offers an explanation as to when and why RL is preferred to other learning methods. RL is particularly powerful in its role as a Markov Decision Process (MDP). MDPs, named after Andrey Markov, provide a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker (agent). MDPs are useful for studying a wide range of optimisation problems solved via DP and RL (Puterman, 1994). An MDP is a discrete time stochastic control process where the process is in a certain state s (at each time step), with certain actions a available to it. The agent then chooses one of these actions, after which the process responds by randomly moving to a new state s'. After moving to this state, the process returns a reward.

The specific action influences the probability of s' being chosen as the next state. This probability is given by the state transition function. The next state s' depends on the current state s and the decision maker's action a, but not on any previous states and actions. The state transitions of an MDP possess the Markov property. RL has the ability to solve the MDP without computing the transition probabilities that are needed in value and policy iteration (Sutton and Barto, 1998a).

RL is arguably better than other learning methods, because it adapts quickly to a changing environment. The reason is the goal-directedness of RL. A learning agent interacts with its environment to achieve a goal. Such an agent must be able to take actions that affect the state and must also have a goal/goals relating to the state of the environment. RL gives a measure of appropriateness of the actions taken and the agent then adjusts its memory to select a better action next time. The Command and Control (C2) operator in the military environment can be thought of as the agent, because he/she does not always have the necessary situational awareness or time to make decisions.

The bulk of this dissertation is concerned with RL, but ultimately we want to apply it to the problem of Weapon Assignment (WA). We must first have a good understanding and background of RL and in order to do this we need to study the different methods and approaches. The theory is mostly from Sutton and Barto (1998a) and is explained by means of examples.

1.4 A road map to the dissertation

The dissertation is divided into two main parts: RL and WA. Most of the work is concerned with RL. The last few chapters are dedicated to the problem of WA in order to demonstrate the workings of RL effectively.

Chapter 2 takes a theoretical approach to explaining RL. We introduce the core components of RL. We discuss the concepts of evaluative feedback, action-value methods, softmax action selection and we look at the difference between evaluation and instruction. After that incremental implementation is discussed, as well as how to track a non-stationary problem and setting optimistic initial values.

We revisit the RL problem where we look at the agent-environment interface, goals and rewards, returns, and a unified notation for episodic and continuing tasks. We introduce the Markov property and MDPs which will be used extensively throughout the dissertation. We then get to value functions, optimal value functions and conclude the chapter with a look at optimality and approximation.

In Chapters 3-5 we discuss the three standard solution methods for the RL problem: Dynamic Programming (DP), Monte Carlo (MC) and Temporal-Difference (TD) learning. In each chapter we start off with a quick background overview before getting to the theory behind the various methods and their underlying algorithms. We also illustrate a few of the algorithms at the hand of examples. In Chapter 6 we look at the first of the ways to unify the elementary solution methods, namely Eligibility Traces.

Chapter 7 signals the start of the second part of the dissertation. Here we do a review of the available literature on Threat Evaluation and Weapon Assignment (TEWA), and in particular, WA. We give a quick overview of Air Defence (AD), in particular, Ground Based Air Defence (GBAD). We discuss the role of the operator and the tactical environment. We briefly look at the detection and identification of threats as well as the Fire Control Operator (FCO), the flight path prediction module, and C2.

In Chapter 8 we discuss the application and implementation of our simplified WA problem, while the results are displayed graphically and discussed in Chapter 9. Chapter 10 concludes the dissertation by reviewing our most important research findings and discussing possible future work.

Chapter 2

Background and theory of Reinforcement Learning

2.1 Introduction

In this chapter we look at the history and the general ideas behind Reinforcement Learning (RL). We consider how an RL problem is defined and introduce the concepts of exploration and exploitation, give examples and applications, as well as point out some limitations. We move on to the history of RL and take a look at where RL fits into the greater context. We present the most important components of the RL problem, such as the interaction between the agent and the environment, and introduce a unified notation for episodic and continuing tasks. We define the Markov property as well as Markov decision processes. After that we discuss evaluative feedback where we consider the n-lever slot machine problem, methods for action selection, incremental implementation, and optimistic initial values.

2.2 The RL route

RL is part of machine learning (a discipline that is concerned with the design and development of algorithms that allow computers to learn from data, and also a branch of Artificial Intelligence (AI) (Russell and Norvig, 2003). RL is intended to be a simple way of representing essential features of the AI problem. RL uses a formal framework defining the interaction between a learning agent and its environment in terms of states, actions and rewards. It is concerned with how an agent should take actions in an environment to maximise some sort of long term reward (Sutton and Barto, 1998*a*). This numerical reward is also known as the *reinforcement signal* (Champandard, 2002). What sets RL apart from other machine learning methods is the fact that the learner is not told which actions to take, but instead

must discover which actions yield the highest reward by trying them all. RL starts with a complete, interactive, goal-seeking agent. All RL agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. The agent has to operate although it does not know everything about the environment. This is particularly useful and important in applications such as Weapon Assignment (WA) where the necessary supervision and rules of engagement may not always be readily available.

In Figure (2.1) we see the loop for a general RL problem. The agent receives information from its environment, usually as a sensory input, which gives information about the current state of the environment (Engelbrecht, 2007). The agent chooses an action and upon execution, receives a reward. This reward can be positive or negative, depending on the appropriateness of the action. A negative reward would be given for a bad action. Depending on the action taken, the environment might change, thereby affecting the states and actions possible in the future.

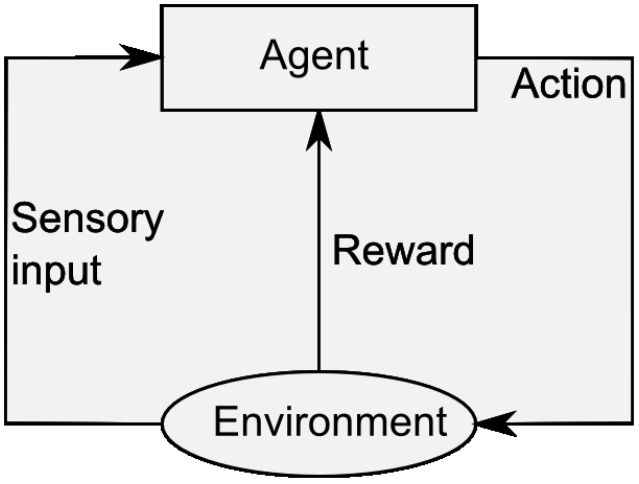


Figure 2.1: The RL loop

The rest of this section is divided into subsections to shed more light on RL. We look at methods for solving applicable RL problems and then move on to the concept of exploration versus exploitation. We name a few examples and applications of RL, and also state the limitations.

2.2.1 Methods and problems

There are many different algorithms that tackle the issue of discovering actions that yield the highest reward. RL is not defined by characterising learning methods, but by a learning problem (Sutton and Barto, 1998*a*). RL algorithms try to find a policy that maps states of the world to the actions the agent should take in those states. In simpler terms, the

agent decides what the best action is to select based on its current state. When this step is repeated, the problem is known as a *Markov Decision Process* (MDP). This will be discussed in more detail in Section 2.6.3.

The most popular type of method for solving the RL problem is to allow the agent to select an action that will maximise the long-term return (and not only the immediate reward). Such algorithms are known to have *infinite horizon* (the terminal time is taken in the limit). In practice, this is done by learning to estimate the value of a particular state. This estimate is adjusted over time by propagating part of the next state's reward. If all the states and all the actions are tried enough times, an optimal policy can be found and the action that maximises the value of the next state is picked (Sutton and Barto, 1998*a*).

2.2.2 Exploration versus exploitation

In uncharted territory an agent must be able to learn from its own experience. The challenge faced is the trade-off between exploration and exploitation. To obtain the highest reward, an RL agent must prefer actions that it has tried in the past and found to be effective. To do even better, it has to try actions that it has not selected before. The agent has to exploit what it already knows in order to obtain reward, but it also has to explore in order to make better decisions in the future. The dilemma we are faced with is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and increasingly favour those that appear to be best. Taking non-optimal actions may get it to the "road less travelled" often enough to learn something new, thereby updating the estimate and enabling better actions in the future.

2.2.3 Examples and applications

The possible applications of RL are many and varied due to the broad nature of the problem specification (Champandard, 2002). In practice, this ranges from controlling a robotic arm to pour cooldrink into a glass, to certain games. Suitable games are traditionally defined as a sequence of decisions and include poker, backgammon, blackjack and chess.

In a chess game the agent is the chess player making his move. The choice is informed both by planning (expecting possible replies and counter replies) and by the agent's judgment of the desirability of certain positions and moves. It involves the interaction between an active agent and its environment, within which the agent tries to achieve a goal despite uncertainty about its environment. The agent's actions are allowed to affect the future state of the environment (*e.g.* the next chess position), thereby affecting the options and opportunities available to the agent at later times. The correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning.

The effects of actions cannot be fully predicted and so the agent must monitor its environment frequently and react appropriately. This involves goals that are specified so that the

agent can judge progress towards its goal based on what it can sense directly. The chess player knows whether or not he wins. The agent can use its experience to improve its performance over time. The chess player refines the intuition he uses to evaluate positions, thereby improving his play. Interaction with the environment is essential for adjusting behaviour to exploit specific features of the task.

Other examples of RL include process control (Hoskins and Himmelblau, 1992), scheduling (Zhang, 1995) and resource allocation (Naidoo, 2008). A very important and classic resource allocation example is the problem of optimally assigning weapons to threats– the *Weapon Assignment* problem that we mentioned earlier.

2.2.4 Limitations

There are many challenges in current RL research. Firstly, if the problem is complex, it can be memory expensive to store values of each state. Solving this involves looking to value approximation techniques, such as *decision trees* or *neural networks*.

Similar behaviours also reappear often, and modularity can be introduced to avoid learning everything all over again. Finally, due to limited perception, it is often impossible to fully determine the current state. This also affects the performance of the algorithm, and much work has been done to compensate for this *Perceptual Aliasing* (different states that appear similar but require different responses) (Champandard, 2002).

2.3 History of Reinforcement Learning

2.3.1 Introduction

According to Sutton and Barto (1998*a*), the history of RL has two main threads which were followed independently before coming together in modern RL. One thread is concerned with learning by trial-and-error and started in the psychology of animal learning. This thread runs through early AI and led to the revival of RL in the early 1980s. The other thread is concerned with the problem of optimal control and its solution using value functions and dynamic programming. Although the two threads were mostly independent, there are a third, less distinct thread, that brought everything together. This thread is concerned with Temporal-Difference (TD) methods (Sutton and Barto, 1998*a*). All three threads came together in the late 1980s to produce the modern field of RL as presented here.

2.3.2 The problem of optimal control

We start off by considering the problem of optimal control. One of the approaches to this problem used the concept of a dynamic system's state and of a value function to define a func-

tional equation, now called the *Bellman equation*. The class of methods for solving optimal control problems by solving this equation came to be known as *Dynamic Programming* (DP) (Bellman, 1957*a*). Bellman (1957*b*) also introduced the discrete stochastic version of the optimal control problem known as *Markov Decision Processes* (MDPs), and Howard (1960) devised the policy iteration method for MDPs. All of these are vital elements underlying the theory and algorithms of modern RL.

In many circles DP is considered to be the only feasible way of solving general stochastic optimal control problems, but it suffers from what Bellman called "the curse of dimensionality", meaning that its computational requirements grow exponentially with the number of state variables. It also requires full knowledge of the system. Despite this requirement of full knowledge and the curse of dimensionality, DP is still viewed as an efficient and widely applicable method.

The solution methods of optimal control, such as DP, are also considered as RL methods.

2.3.3 Trial-and-error learning

The other major thread leading to the modern field of RL, the one centered around the idea of trial-and-error learning, began in psychology where "reinforcement" theories of learning are common. The two most important aspects of what is meant by trial-and-error learning is firstly that it is *selectional* rather than instructional (it involves trying alternatives and selecting among them by comparing their consequences). Secondly, it is *associative*, meaning that the alternatives found by selection are associated with particular situations. Supervised learning is associative, but not selectional. Trial-and-error is an elementary way of combining search and memory: *search* in the form of trying and selecting among many actions in each situation, and *memory* in the form of remembering what actions worked best, associating them with the situations in which they worked best. Combining search and memory in this way is essential to RL.

In early AI, several researchers began to explore trial-and-error learning as an engineering principle (Minsky, 1954). In the 1960s we find the terms "reinforcement" and "RL" being widely used in the engineering literature for the first time. Particularly influential was Minsky's paper "Steps Toward Artificial Intelligence" (Minsky, 1961), which discussed several issues relevant to RL, including the credit-assignment problem.

2.3.4 Temporal-Difference learning

The third thread in the history of RL is the one concerning Temporal-Difference (TD)-learning (Sutton and Barto, 1998*a*). TD-learning methods are driven by the difference between temporally successive estimates of the same quantity. Predictions are updated to match other more accurate predictions. This thread is less distinct than the other two, but

has played a particularly important role in the field, mainly because TD methods seem to be unique to RL (Sutton and Barto, 1998*a*).

The origins of TD-learning can be traced back to animal learning psychology, in particular to the concept of *secondary reinforcers*. A secondary reinforcer is a stimulus that has been paired with a primary reinforcer and, as a result, has come to take on similar reinforcing properties. An example of a secondary reinforcer would be the sound from a clicker, as used in clicker training. The sound of the clicker has been associated with praise or treats, and thus the sound of the clicker may function as a reinforcer. Minsky (1954) may have been the first to realize that this psychological principle could be important for artificial learning systems.

2.3.5 Integration of the threads

The TD and optimal control threads were fully integrated in 1989 with Chris Watkins's development of Q-learning. This work extended and integrated previous work in all three threads of RL research. Werbos (1987) also contributed to this by arguing for the convergence of trial-and-error learning and DP. By the time of Watkins's work there had been remarkable growth in RL research, mainly in the machine learning subfield of AI. The success of Gerry Tesauro's backgammon playing program, *TD-Gammon* (Tesauro, 1995), brought further attention to the field.

2.4 The position of RL in the greater context

According to sources (Sutton and Barto (1998*a*), Gasser (2009), Hu *et al.* (2007)), RL is not supervised learning. Although RL is a "trial-and-error" approach and there is no supervisor present, the algorithm learns from interaction and not from examples. It is important to note that there are not always examples available for the WA problem. RL allows the agent to learn its behaviour based on feedback from the environment (Champandard, 2002). This behaviour can be learned and adopted permanently, or it may be adapted continuously.

This automated learning scheme implies that there is little need for a human expert who knows about the domain of application. Much less time will be spent designing a solution, since there is no need for hand-crafting complex sets of rules, and all that is required is someone familiar with RL. This would be very useful in a WA context as we mentioned at the start of the chapter.

Learning algorithms fall into three groups with respect to the sort of feedback that the agent has access to. The one extreme is *supervised learning*: learning takes place under matched input and output patterns or examples. The environment tells the agent what its response should be (Gasser, 2009) and the agent then compares its actual response to the target and adjusts its internal memory in such a way that it is more likely to produce the appropriate

response the next time it receives the same input. For instance, if an agent is provided with many pictures that he is told contain trains, the agent learns to recognise a train.

On the other extreme is *unsupervised learning*: the agent receives no feedback from the world at all. Patterns are learned by providing input, but in the absence of specific outputs. Unsupervised learning is based on the similarities and differences among the input patterns. For example, when commuting from home to work, a person might be able to distinguish between "good traffic days" and "bad traffic days", without ever being given examples of either of the two.

RL is much closer to supervised than unsupervised learning. The agent receives feedback about the appropriateness of its response. For correct responses, RL resembles supervised learning: in both cases, the learner receives information that what it does is appropriate. However, the two forms of learning differ significantly in situations in which the learner's behaviour is in some way inappropriate. In these situations, supervised learning lets the agent know exactly what it should have done, whereas RL only says that the behaviour was inappropriate and (usually) how inappropriate it was. This can be thought of as the game you used to play when you were little where you would direct your friend towards an object by saying "hot" or "cold", *et cetera*. You did not tell your friend he or she was "wrong" or "right" for walking in a certain direction, you gave them a measure of how appropriate their actions were.

In practice, RL is much more common than supervised learning. A teacher (especially in WA where time is of the essence) is not always available who can say what should have been done when a mistake is made, and even when such a teacher is available, it is rare that the learner can interpret the teacher's feedback correctly. Consider an animal that has to learn some aspects of how to walk. It tries out various movements; some work, it moves forward and is rewarded. Others fail, it stumbles or falls down and is punished with pain.

2.5 The components of RL

This section is divided into several smaller parts consisting of certain components of RL. These components are the agent-environment interface, the policy, returns, value functions, optimal value functions and a model of the environment. Of these, the two main components of an RL problem are the learner (agent), and the environment. Other important components are the policy, the value function and returns. A model of the environment is optional. By "model" we mean anything that an agent can use to predict how the environment will respond to its actions.

2.5.1 The agent-environment interface

The agent selects actions and the environment responds to those actions and presents the agent with new situations. They interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$ and at each of these steps, the agent receives some representation of the environment. The time steps do not necessarily refer to fixed intervals of real time, they can refer to successive stages of decision making and acting. The agent has to operate despite considerable uncertainty about the environment it faces. It senses the state of the environment it finds itself in and can take actions that affect the state. The effects of the actions cannot be fully predicted and the agent must monitor its environment frequently and react accordingly. Sometimes the goals are explicit in the sense that the agent can judge progress toward its goal based on what it can "see". The agent uses its experience to improve its performance over time. The sequence of events can be thought of as

$$s_t \rightarrow a_t \rightarrow s_{t+1} \rightarrow r_{t+1}.$$

The agent is in a given state, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states. On that basis it selects an action $a_t \in \mathcal{A}(s_t)$ where $\mathcal{A}(s_t)$ is the set of actions available in state s_t. One time step later, the agent finds itself in a new state, s_{t+1}, and receives a numerical reward, $r_{t+1} \in \Re$.

At each time step the agent implements the *policy*, which is a mapping from states to actions. The policy is denoted π_t, where $\pi_t(s, a)$ is the probability of taking action $a_t = a$ when in state $s_t = s$. RL methods specify how the agent changes its policy according to its experience.

When speaking about the "agent and the environment", it is natural to think of a situation where a person is standing in, say, a garden. There is a definite physical boundary between the person and the garden, but in RL this boundary is not always so definite or clear. The general rule is that anything that cannot be changed by the agent is considered to be outside of it and thus part of the environment. In some cases the agent may know everything about how its environment works (like a game of chess) and still face a difficult RL task. The agent-environment boundary represents the limit of the agent's control, not of its knowledge.

The reward computation is considered to be outside of the agent's control, because it defines the task at hand. The reason for this is that the problem would probably not end up the way it was intended to if the agent were to choose its own rewards. If you teach a dog to do tricks you would give it treats when it performs the tricks correctly. If the dog were to choose its own rewards it would probably devour all the treats and do none of the tricks!

2.5.2 The Policy

The deterministic policy $\pi(s)$ defines the agent's way of behaving at a given time and is a mapping from states to actions. Sometimes it can be a simple function or lookup table, other times it can be a search process. The policy is the core of an agent as it determines its behaviour.

2.5.3 Returns

As mentioned previously, we want to maximise the long term reward. This long term reward accumulation is called the *expected return*, R_t. In the simplest case the return for an episodic problem is the sum of the rewards:

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T,$$

where T is a final time step.

This makes sense in applications where there is a natural idea of a final time step, such as when the agent-environment interaction breaks naturally into *episodes*. Examples of episodes are plays of a game, trips through a maze– any type of repeated interactions. Each episode ends in a special state called the terminal state, after which it is reset to a starting state.

Many times the agent-environment interaction does not divide naturally into identifiable episodes, but goes on continually. This is called *continuing tasks*. The sum-of-rewards mentioned above would be problematic here because the final time step would be $T = \infty$, seeing that the interaction goes on without limit, and the return could also be infinite.

To overcome this problem of an infinite return, we introduce *discounting*. With discounting the agent tries to select actions so that the sum of the discounted rewards received over the future is maximised. It chooses a_t to maximise the expected discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where γ is a *discount parameter*, $0 \leq \gamma \leq 1$.

Discounting is a well-known term in finance where we work with the time value of money. In RL, just as in finance, the discount rate determines the current value of future rewards. A reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence is bounded. If $\gamma = 0$, the agent is shortsighted and only concerned with maximising immediate rewards. As γ approaches 1 the agent becomes more farsighted by taking future rewards more strongly into account.

2.5.4 Value functions

Value functions are functions of states (or state-actions pairs) that tell the agent how good it is for him to be in a given state. "How good" is defined in terms of future rewards that the agent can expect to receive, which in turn depend on what actions it will take.

A value function is defined with respect to a particular policy, where the policy π is a mapping from each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$ to the probability $\pi(s, a)$ of taking action a when in state s. There are two types of value functions, namely *state-value functions* and *action-value functions*. The state-value of a state s under a policy π, denoted $V^\pi(s)$, is the expected return when starting in s and following π thereafter. For Markov Decision Processes (MDPs, Section 2.6.3), we can define $V^\pi(s)$ as:

$$V^\pi(s) = \mathrm{E}_\pi \{R_t | s_t = s\} = \mathrm{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\},$$

where $\mathrm{E}_\pi\{\}$ is the expected value given that the agent follows policy π.

We define the value of taking action a in state s under a policy π, denoted $Q^\pi(s, a)$, as the expected return starting from s, taking action a, and from then on following policy π:

$$Q^\pi(s, a) = \mathrm{E}_\pi \{R_t | s_t = s, a_t = a\} = \mathrm{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}.$$

Value functions satisfy particular recursive relationships. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^\pi(s')],$$

and this is called the *Bellman equation* for V^π. This equation states the relationship between the value of a state and the values of previously encountered states. The agent could take any action in a certain subset of the action space. From each of these selections, the environment responds with one of several next states s' along with a reward r. The Bellman equation averages over all the state-action possibilities weighting each by its probability of occurring. The value of the start state must equal the discounted value of the expected next state, plus the reward expected along the way.

2.5.5 Optimal value functions

To solve a RL task, we must find a policy that maximises the reward over the long run. We define one policy π to be better than another policy π' if its expected return is greater or equal to that of π' for all states. This is written as $\pi \geq \pi'$ if and only if $V^\pi(s) \geq V^{\pi'}(s)\ \forall s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies and

this is called the *optimal policy*. We denote the optimal policy (as do we the optimal value functions) by a "$*$", hence π^*. If there is more than one optimal policy, they all share the same optimal state-value function, V^*, and the same optimal action-value function, Q^*. The optimal policy yields the highest expected return, thus, for all states and actions:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ &\text{and} \\ Q^*(s,a) &= \max_\pi Q^\pi(s,a). \end{aligned}$$

V^* is the value function for a policy, therefore it must satisfy the self-consistency condition given by the Bellman equation for state values. The consistency condition for V^* can be written in a special form without reference to any specific policy. This is the *Bellman optimality equation* for V^*. The value of a state s under an optimal policy π^* must equal the expected return for the best action a from that state:

$$V^*(s) = \max_a \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')].$$

The corresponding Bellman optimality equation for Q^* is:

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')].$$

It is relatively simple to determine an optimal policy once V^* is known. For each state s, there is one or more actions at which the maximum occurs in the Bellman optimality equation. Once V^* is known, the actions that appear best after a one-step search are the optimal actions. Any policy that is greedy with respect to the optimal value function V^* is an optimal policy.

Having Q^* makes choosing optimal actions even easier. With Q^*, the agent does not even have to do a one-step lookahead search: for any state s, it can simply find any action that maximises $Q^*(s,a)$. The action-value function provides the optimal expected long-term return as a value that is immediately available for each state-action pair.

2.5.6 Model of the environment

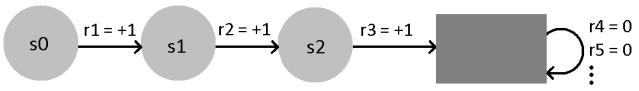
A model mimics the behaviour of the environment. Given a state and an action, a model produces a prediction of the resultant next state and next reward. Models are used for deciding on a plan of action by considering possible future situations before they are actually experienced. A model can be *stochastic* or a *distribution model*.

2.6 More RL concepts

2.6.1 Unifying episodic and continuing tasks

When we discuss episodic tasks we almost never distinguish between different episodes, but rather consider a particular single episode, or state something that is true for all episodes.

When working with episodic tasks, we consider the sum over a finite number of terms and when working with continuing tasks, we consider the sum over an infinite number of terms. These two cases can be unified by considering that when the episode terminates, a special absorbing state is entered. This terminal state transitions only to itself and generates only rewards of zero. For example:



Summing these rewards, we get the same return whether we sum over the first T rewards (here $T = 3$) or over the full infinite sequence. This is true even if discounting is introduced. We can now define the return generally as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots,$$

including the possibility that $\gamma = 1$ if the sum remains finite. We can also write the returns as

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1},$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

2.6.2 The Markov property

The agent makes its decisions based on a current state. We assume that the state is given by the environment, and by "state" we mean the information available to the agent. We need to decide which action to take as a function of the state.

The ideal is to have a state signal that keeps a record of past sensations, but in such a way that not every single event is retained. This means that the immediate sensations are kept, but never the complete history of all past sensations. An example of this would be a snapshot of a chess game. You do not need to know what actions led to the current view of the board, you are only interested in what the current board setup is.

A state signal that succeeds in keeping all relevant information is said to have the Markov property. The chess board mentioned above would contain in it the current configuration of

all the pieces on the board. This serves as a Markov state because it summarises everything important about the history of choices that led to it, without keeping a full record. Another example is assigning weapons to a threat. We do not need to know which actions or states led to the current position, all we care about is where the threat is and what the shooting order is in that given state.

Let us look at the difference between having the Markov property and not having it. A general environment might respond at time $t+1$ to an action taken at time t. We say the responses are *conditionally independent*. The equation $p(x|a,b) = p(x|a)$ means x is independent of b if and only if a is known. So $p(x_{t+1}|x_t,\ldots,x_0) = p(x_{t+1}|x_t)$ means that, if x_t is known, then x_{t+1} is independent of $x_{t-1},\ldots,x_0$.

If a response were to depend on everything that has happened previously, the dynamics can be defined only by specifying the complete probability distribution:

$$\Pr\{s_{t+1} = s', r_{t+1} = r|s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\},$$

for all s', r, and all possible values of the past events: $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$. If the state signal has the Markov property, then by definition the environment's response at time $t+1$ depends only on the previous state and action representations at time t, in which case the environment's dynamics can be defined by specifying only $\Pr\{s_{t+1} = s', r_{t+1} = r|s_t, a_t\}$, for all s', r, s_t and a_t. A state signal has the Markov property and is a Markov state if and only if the previous two probability distributions are equal for all s', r and histories $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$.

If an environment has the Markov property, we can predict the next state and expected reward given the current state and action. By iterating this, we can predict all future states and expected rewards from knowledge as well as would be predicted given the complete history up to the current time. The same goes for choosing actions. Choosing actions as a function of a Markov state is just as good as choosing actions as a function of complete histories.

All of the theory presented in this dissertation assumes Markov state signals.

2.6.3 Markov decision processes

An RL task that satisfies the Markov property is called a *Markov Decision Process*, or MDP. If the state and action spaces are finite, then it is called a finite MDP.

A specific finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state and action, s and a, the probability of each possible next state s' is $\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s'|s_t = s, a_t = a\}$. These quantities are called *transition probabilities*.

Given any current state and action, s and a, together with any next state s', the expected

value of the next reward is $\mathcal{R}^a_{ss'} = \mathrm{E}\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$. These quantities completely specify the most important aspects of the dynamics of a finite MDP.

2.7 Evaluative feedback

RL does not tell the agent if the correct action was taken, but evaluates the actions taken and tells the agent how appropriate it was. This creates the need to explore. Evaluative feedback indicates how good the action taken is, but not whether it is the best or worst action possible.

The rest of this section is divided into subsections consisting of a slot machine problem, action-value methods, softmax action selections, incremental implementation and optimistic initial values.

2.7.1 A slot machine problem

Assume for the moment that you are faced with a slot machine with n levers and that you have to choose repeatedly among the n different actions (each having a different expected reward). After each choice you receive a reward that depends on the action you selected. The goal is to maximise the expected total reward over a time period, say 1,000 action selections. Each action selection is called a play because each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs. Through repeated plays you want to obtain the highest winnings by choosing the best levers.

The value of an action is the expected return given that that action is selected. If you knew which lever yields the highest winnings, you would always choose that lever. This is one of the core ideas in RL: you may have estimates, but you do not know with certainty what is best.

If you keep track of the action-value estimates ($Q^\pi(s, a)$), then there is always at least one action whose estimated value is greatest. This action is called a *greedy* action. If you select a greedy action, you are exploiting what you already know about the action values; you know which actions yield high rewards. If you select one of the non-greedy actions, it means that you are not using your current knowledge of the action values, but exploring in the hope of finding some action that is better than the one suggested by the policy. This enables you to improve your estimate of the non-greedy action's value.

One of the great challenges of RL is to balance the need for exploration and exploitation. Choosing actions that you know yield high rewards (exploitation) might maximise the expected return of one play, but choosing other actions that you do not know much about (exploring) might produce greater total return in the long run.

Suppose that the greedy action's value is estimated and several other actions are estimated to be nearly as good. At least one of the other actions might be better than the greedy action, but you do not know which one. Exploring the non-greedy actions might discover which of them are better in the long run. A lower reward might be obtained in the short run during exploration, but the reward is higher in the long run seeing that you have discovered better actions and can now exploit them. As we explained in Section 2.2.2, it is not possible to only exploit or only explore, there is a need to balance the two.

2.7.2 Action-value methods

In this subsection we look at simple methods for estimating the action values and then using these estimates to make action selection decisions.

Let $Q^*(s,a)$ be the actual value of the state-action pair (s,a) and $Q_t(s,a)$ the estimated value at the t^{th} play. As stated above, the true value of a state-action pair is the mean return received when that action is selected. When you hear the word "mean", you immediately think "average" and this is exactly how the *sample-average method* for estimating action values works. You average the returns actually received when the action was selected. If at the t^{th} play action a is chosen k_a times prior to t, yielding rewards $r_1, r_2, \ldots, r_{k_a}$, then its value is estimated to be

$$Q_t(s,a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}.$$

If $k_a = 0$, we would divide by zero, thus we define $Q_t(s,a)$ instead as some default value, *e.g.* $Q_0(s,a) = 0$. As $k_a \to \infty$, $Q_t(s,a)$ converges to $Q^*(s,a)$.

The simplest action selection rule is to choose the action (or one of the actions if there are more than one) with the highest estimated action value. Referring to the slot machine problem, this will mean choosing the lever on play t that yielded the highest winnings. Symbolically, this means choosing one of the greedy actions a^* for which

$$Q_t(s,a^*) = \max_a Q_t(s,a).$$

This method always exploits current knowledge. A simple alternative is to behave greedily most of the time, but every once in a while, with a small probability ϵ, select a random action, independently of the action-value estimates. These methods are called *ϵ-greedy methods*. The advantage of these methods is in the limit as the number of plays increases. Every action will be sampled an infinite number of times, guaranteeing that $k_a \to \infty$ for all a, and thus ensuring that all the $Q_t(s,a)$ converge to $Q^*(s,a)$.

Greedy methods improve slightly faster than ϵ-greedy methods, but perform much worse in the long run because greedy methods get into the habit of choosing suboptimal actions by only exploiting and never exploring. The ϵ-greedy methods perform better in the long run because they continue to explore, thereby improving their chances of recognising the

optimal action. It is important to note that the advantage of ϵ-greedy over greedy methods is task dependent. Sometimes it takes more exploration to find the optimal action, and greedy methods would fare even worse compared to ϵ-greedy methods.

2.7.3 Softmax action selection

The disadvantage of using ϵ-greedy action selection is that when it explores it chooses equally among all actions– the worst action is chosen with the same probability as the best action.

One way to overcome this is to give the greedy action the highest selection probability, but to rank and weight all the others according to their value estimates. This is called *softmax* action selection rules (Sutton and Barto, 1998a). Action a on the t^{th} play is chosen with probability

$$\frac{e^{Q_t(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(s,b)/\tau}},$$

where τ is a positive parameter called the *temperature*. High temperatures cause the actions to almost all have the same probability of being selected. As $\tau \to 0$, softmax action selection becomes the same as greedy action selection.

2.7.4 Incremental implementation

We now see how easy it is to devise incremental update formulas for computing averages required to process each new reward without having to store it. For some action, let Q_k denote the average of its first k rewards (not $Q_k(s,a)$ which is the average for action a at the k^{th} play). Given this average and a $(k+1)$st reward r_{k+1}, the average of all $k+1$ rewards can be computed by

$$Q_{k+1} = Q_k + \frac{1}{k+1}[r_{k+1} - Q_k]. \tag{2.1}$$

The update rule is of the form

$$\text{newEstimate} \leftarrow \text{oldEstimate} + \text{Stepsize}[\text{Target} - \text{oldEstimate}],$$

where "Target - oldEstimate" is the error in the estimate. The error is reduced by taking a step towards the target.

2.7.5 Optimistic initial values

All the methods discussed so far depend in some way on the initial action-value estimates, $Q_0(s,a)$ and this makes the methods biased by their initial estimates. For the sample-average methods, the bias disappears when all actions have been selected at least once. For methods with constant stepsize parameter α, the bias is permanent, though decreasing over

time. The disadvantage is that the initial estimates become a set of parameters that must be picked by the environment, if only to set them all to zero.

Initial action values can also be used as a way of encouraging exploration. Instead of setting the initial action values to zero, say we set them all to +3. This is an optimistic estimate, since the estimated action-values are never greater than one. This optimism encourages action-value methods to explore. No matter which actions the method initially selects, the reward is always going to be less than the +3 starting estimate. The agent is then disappointed with the reward it receives and switches to other actions, exploring to see if it can find something better. The result is that all actions are tried several times before the value estimates converge.

2.8 Concluding remarks

In this chapter we discussed the history and theory behind RL. We saw that the RL agent learns from interaction how to behave in order to achieve a goal and that the agent and its environment interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the actions are the choices made by the agent; the states are the basis for making the choices; and the rewards are the basis for evaluating the choices. Everything inside the agent is completely known and controllable by the agent, while everything outside is incompletely controllable, but may or may not be completely known.

A policy is a stochastic rule by which the agent selects actions as a function of states and the return is the function of future rewards that the agent seeks to maximise. Policies have several definitions depending on the nature of the task and whether one wishes to discount delayed rewards. The undiscounted formulation is appropriate for episodic tasks and the discounted formulation is appropriate for continuing tasks.

The environment satisfies the Markov property if its state signal compactly summarises the past without degrading the ability to predict the future. If the Markov property holds, then the environment defines an MDP.

A policy's value function assigns to each state the expected return from that state, given that the agent uses the policy. Optimal value functions assign to each state the largest expected return achievable by any policy. Any policy that is greedy with respect to the optimal value functions must be an optimal policy. The Bellman optimality equations are special consistency conditions that the optimal value functions must satisfy and that can be solved for the optimal value functions, from which an optimal policy can be determined.

Chapter 3

Dynamic Programming

3.1 Introduction

In this chapter we look at the first of three standard methods for solving the Reinforcement Learning (RL) problem. We start off with the theory and general idea behind Dynamic Programming (DP) and give a few examples. We then move on to policy evaluation and policy improvement. After that we combine these two computations, obtaining policy iteration and value iteration. We then look at asynchronous DP, generalised policy iteration and the efficiency of DP.

3.2 Theory of Dynamic Programming

Dynamic Programming (DP) is a collection of algorithms that solve problems by breaking them down into simpler sub-problems (Sutton and Barto, 1998*a*). In RL these algorithms can be used to compute optimal policies. DP methods require a complete model of the environment, usually as a Markov Decision Process (MDP).

Let us assume the environment is a finite MDP, where *finite* refers to the fact that both the state set $\mathcal{S}$ and action set $\mathcal{A}(s)$ are finite. We assume further that the environment's dynamics are given by a set of transition probabilities as defined in Section 2.6.3,

$$\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}.$$

This is the probability of being in state s' at time $t+1$ given that the state at the previous time step was s and the action taken a. The expected immediate reward is

$$\mathcal{R}^a_{ss'} = \mathrm{E}\{r_{t+1} | a_t = a, s_t = s, s_{t+1} = s'\}$$

for all states $s \in \mathcal{S}$, actions $a \in \mathcal{A}(s)$ and next states $s' \in \mathcal{S}^+$, where $\mathcal{S}^+$ is the state space $\mathcal{S}$ plus a terminal state if the problem is episodic (Sutton and Barto, 1998*a*). This expected immediate reward gives the expected value of receiving reward r at time $t+1$ given that the agent was in state s at time t, took action a and observed s' as the next state at time $t+1$.

The main idea of DP is to use value functions to search for good policies. Once the optimal value functions, V^* or Q^*, are available, the optimal policies can be obtained. These optimal value functions satisfy the Bellman optimality equations,

$$\begin{aligned} V^*(s) &= \max_a E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V^*(s')] \end{aligned} \tag{3.1}$$

or

$$\begin{aligned} Q^*(s,a) &= E\{r_{t+1} + \gamma \max_{a'}\, Q^*(s_{t+1}, a') | s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, \max_{a'}\, Q^*(s', a')], \end{aligned} \tag{3.2}$$

for all states $s \in \mathcal{S}$, actions $a \in \mathcal{A}(s)$ and next states $s' \in \mathcal{S}^+$ and where γ is the discount parameter as introduced in Chapter 2. As seen in the next section, DP algorithms are formed by turning Bellman equations into assignments, which are update rules for improving approximations of the value functions.

3.3 Evaluating the policy

Suppose we have some policy π that tells us which action a to choose in state s. We want to know how good it is to choose that specific action; we want to know what the *value* of the choice is. The way we evaluate this policy is by finding the value function V^π of the given policy (henceforth called *policy evaluation*) (Sutton and Barto, 1998*a*). The value function V^π is defined as:

$$\begin{aligned} V^\pi(s) &= E_\pi\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t = s\} \\ &= E_\pi\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} \\ &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V^\pi(s')], \end{aligned} \tag{3.3}$$

where $\pi(s,a)$ is the probability of taking action a in state s by following policy π.

V^π is guaranteed to exist and is unique as long as either $\gamma < 1$ or if we are guaranteed eventual termination from all states under the policy π. If the environment's dynamics are completely known, then V^π is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns. For our purposes, an iterative solution is the method of choice. The reason for this is that the number of states can be high, *i.e.* the linear system can be very large.

The rest of this section will be devoted to iterative policy evaluation.

3.3.1 Iterative policy evaluation

Suppose we have a sequence of value functions, $V_0, V_1, V_2, \ldots$, each computed at a later time step than the previous one. The initial approximation V_0 is given an arbitrary value, usually zero. The value of the terminal state is also made zero. Each consecutive approximation is calculated by using the Bellman equation for V^π as an update rule:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V_k(s')], \tag{3.4}$$

for all states $s \in \mathcal{S}$. To compute the value at a certain state, a backup operation, called a *sweep*, is applied to each state. The function of the sweep is to update the current estimate of the value function. This algorithm is called *policy evaluation*.

To produce the new approximation $V_{k+1}(s)$, the update rule in equation (3.4) is applied to the old approximation $V_k(s)$. This old value $V_k(s)$ is then replaced by a new value $V_{k+1}(s)$ obtained from $V_k(s)$ and the expected immediate rewards. The replacement is done along all the transitions possible in one step under the current policy. This is called a *full backup*. Each iteration backs up the value of every state once to produce the new approximate value function $V_{k+1}(s)$. All the backups done in DP are full backups because they are based on all possible next states rather than on a sample next state.

In the current study, the values are updated in place, with each new backed-up value immediately overwriting the old one. Depending on the order in which the states are backed up, new values are sometimes used instead of old ones on the right hand side of equation (3.4). It has the advantage that new data is used as soon as it is available. Also, it is not necessary to maintain a record array, which is congruent with the fact that it is a Markov Process.

Although iterative policy evaluation converges in the limit, it must be stopped before that in practice, because convergence may take too long. A typical stopping condition is to test the quantity $\max_a |V_{k+1}(s) - V_k(s)|$ each time and stop when it is smaller than a certain θ, where θ is usually much smaller than one. Algorithm 1 gives a complete algorithm for iterative policy evaluation with this stopping criterion.

Algorithm 1 Iterative policy evaluation

Initialise π, the policy to be evaluated
Initialise $V(s) = 0$, for all $s \in \mathcal{S}^+$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$
Output $V \approx V^\pi$

3.3.2 Gridworld example (Sutton and Barto, 1998*a*)

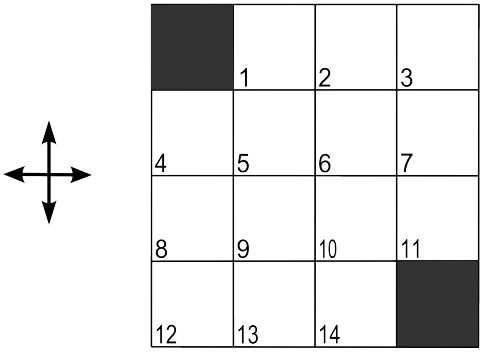


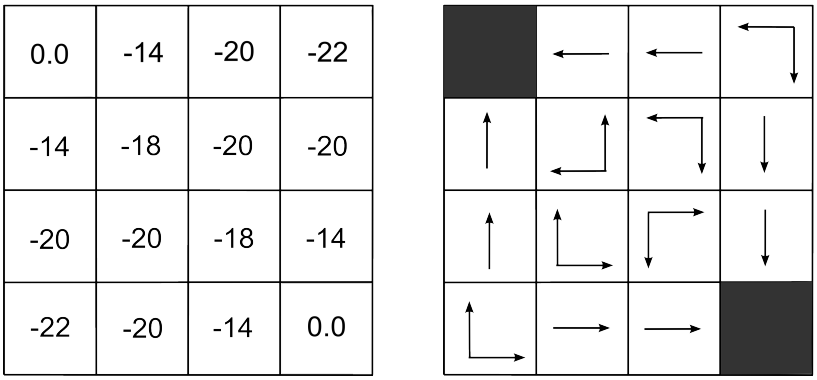
Figure 3.1: 4×4 gridworld example

Suppose we have a grid we wish an agent to walk through, as seen in Figure (3.1). The agent can start in any state and take any one of four actions: up, down, right and left, where all four directions have the same probability of being chosen. If the agent selects an action that will take him off the grid, it leaves his position unchanged. The goal is to reach one of the shaded cells.

The non-terminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$ and the four actions possible in each state are $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$ as already mentioned. These actions cause the resultant state transitions, except actions that take the agent off the grid. For instance, $\mathcal{P}^{\texttt{right}}_{2,3} = 1$, $\mathcal{P}^{\texttt{down}}_{5,8} = 0$, and $\mathcal{P}^{\texttt{left}}_{12,12} = 1$. This is an undiscounted, episodic task. The terminal state is shaded in the figure and is actually one state even though it is shown to be two.

The expected reward function is $\mathcal{R}^a_{ss'} = -1$ for all states s, s' and actions a (all transitions) until the terminal state is reached. This is done to reach the terminal state in as few steps as possible. Suppose further that the agent follows the equiprobable random policy (all the actions have the same probability of occurring) and that the initial value function was initialised as zero. Figure (3.2)(a) shows the value function V_k computed by iterative policy

evaluation. The final estimate is V^π, as shown in figure (a), which gives for each state the negative of the expected number of steps from that state until termination, which occurs after a number of episodes. Figure (3.2)(b) shows the optimal policy to follow in order to obtain the values in figure (a). Notice the high negative values in the middle of the grid; these are the worst states to be in to achieve termination.



(a) Optimal value function (b) Optimal policy

Figure 3.2: Convergence of iterative policy evaluation

3.4 Improving the policy

In the previous section we saw that computing the value function for a policy assists in finding better policies. If we have an arbitrary policy and we calculate the value function, a measure is obtained of how good the policy is. Suppose that the value function V^π has been determined for an arbitrary deterministic policy π and that we wish to improve this policy π based on the results received from V^π. We already know how good the current action selection is, so now we want to know for some state s, should we change the policy to choose a new action a that is not in the current policy, in other words, $a \neq \pi(s)$?

The value of taking action a in state s and then following the current policy π, is

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V^\pi(s')] \end{aligned} \tag{3.5}$$

The question to consider is whether this is greater than or less than $V^\pi(s)$. If it is greater you would expect it to be better to select a every time s is encountered, and that this new and improved policy would be better. This being true is a special case of a general result called the *policy improvement theorem*. The reader is referred to Appendix A.1 for a brief derivation of this theorem.

We now consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $Q^\pi(s,a)$. In other words, we consider the new greedy policy π' given by

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V^\pi(s')]. \end{aligned} \tag{3.6}$$

The greedy policy takes the action that looks best in the short term according to V^π. The greedy policy is designed to meet the conditions of the policy improvement theorem, so we know that it is at least as good as the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

If policy π' is just as good as policy π, then $V^\pi = V^{\pi'}$ and it follows that

$$V^{\pi'}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V^{\pi'}(s')]. \tag{3.7}$$

This is just the Bellman optimality equation, and thus $V^{\pi'} = V^*$, and π' is an optimal policy, but we said that π' is just as good as π, thus both π and π' are optimal policies. From this we can conclude that policy improvement gives a strictly better policy, except when the original policy is already optimal.

In general, a stochastic policy π specifies probabilities $\pi(s,a)$ for taking each action a in each state s. The equation for policy improvement is:

$$Q^\pi(s,\pi'(s)) = \sum_a \pi'(s,a) Q^\pi(s,a). \tag{3.8}$$

3.5 Iterating the policy

Once we have improved our policy π using V^π to obtain a better policy π', we can obtain an even better policy by computing $V^{\pi'}$ and then using that to obtain π'' *et cetera*. We obtain a sequence of improving policies and value functions, each one better than the previous one:

$$\pi \to V^\pi \to \pi' \to V^{\pi'} \to \pi'' \to \dots.$$

A finite MDP has only a finite number of policies, therefore this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This approach of finding an optimal policy is called *policy iteration*, as described by Algorithm 2. Note that each policy evaluation is started with the value function for the previous policy; V^π is used to obtain π' and $V^{\pi'}$ is used to obtain π''. This typically results in a

greater increase in the speed of convergence of policy evaluation. Each iteration of policy iteration involves policy evaluation.

Algorithm 2 Policy iteration algorithm

1. Initialisation
$V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy evaluation
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s'} \mathcal{P}^{\pi(s)}_{ss'}[\mathcal{R}^{\pi(s)}_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$

3. Policy improvement
policy-stable $\leftarrow$ true
for each $s \in \mathcal{S}$ **do**
  $b \leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
  If $b \neq \pi(s)$, then policy-stable $\leftarrow$ false
end for
If policy-stable, then stop, else go to step 2

3.6 Value iteration

Policy iteration can become computationally expensive due to multiple sweeps and to truncate this without losing the guarantee of policy iteration, we use *value iteration*. This is when the policy evaluation is stopped after just one backup of each state.

We can rewrite value iteration as a backup operation that combines the policy improvement and truncated policy evaluation steps:

$$V_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V_k(s')], \tag{3.9}$$

for all states $s \in \mathcal{S}$. When compared to equation (3.4), we can see the difference between hard and soft decisions.

If we refer to the Bellman optimality equations (3.1) and (3.2), we see that value iteration is simply the Bellman optimality equation turned into an update rule. The value iteration backup is identical to the policy evaluation backup, except that this backup requires the maximum to be taken over all actions.

Like with policy evaluation, we stop once the value function changes by only a small amount in a sweep. Algorithm 3 shows a complete algorithm for computing value iteration.

Algorithm 3 Value iteration algorithm

Initialise V arbitrarily, *i.e.* $V(s) = 0$ for all $s \in \mathcal{S}^+$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$

Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$

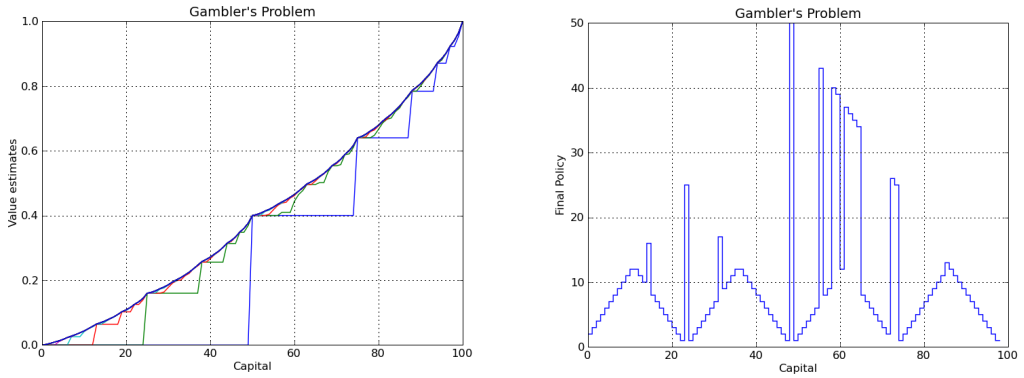
In each of its sweeps, value iteration combines a sweep of policy evaluation and a sweep of policy improvement. To achieve faster convergence, multiple policy evaluation sweeps can be inserted between each policy improvement sweep.

3.6.1 Gambler's problem

Next we consider the gambler's problem. A gambler has the opportunity to place bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip. If the coin comes up tails, he loses his stake. The game ends when the gambler either wins by reaching his goal of $R100$, or loses by running out of money. On each coin flip the gambler must decide how much of his capital to stake, in an integer number of rands. The problem can be formulated as an undiscounted, episodic, finite MDP (Sutton and Barto, 1998*a*).

The states are the gambler's capital, $s \in \{1, 2, \ldots, 99\}$. The actions are the stakes, $a \in \{1, 2, \ldots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, then the reward is $+1$. The state-value function gives the probability of winning from each state. The policy is a mapping from levels of capital to stakes and the optimal policy maximises the probability of reaching the goal. Let p denote the probability of the coin coming up heads. If p is known, then the problem can be solved, for instance, by value iteration. Figure (3.3)(a) shows the change in the value function over successive sweeps of value iteration. Figure (3.3)(b) shows the final policy found, for the case of $p = 0.4$.

The reason for the odd shape of the policy and the outliers is because there are potentially many optimal policies whereas the optimal values are unique. The policy obtained often has



(a) Value functions found (b) Final policy

Figure 3.3: Solution to the gambler's problem for $p = 0.4$

to do with things like numerical resolution or the order in which the program breaks ties for maximal action value. Two numbers may seem to be equal when rounding to, say, the fifth decimal, but when more decimals are inspected it becomes clear that these two numbers are in fact not the same (Email, 1999).

3.7 Asynchronous dynamic programming

DP methods discussed so far involve several sweeps over the entire state set of the MDP. If the state set is very large, then even a single sweep can be very expensive.

Asynchronous DP is an in-place iterative DP algorithm that assumes we are updating one state at a time, after which we update the entire value function (Powell, 2007). In asynchronous DP we could ensure that we sample all states infinitely often by choosing states at random. The term "asynchronous" is based on parallel implementations where updates are not synchronised; that is, not occurring at predetermined or regular intervals. The updates are made intermittently rather than in a steady stream.

These algorithms backup the values of states in any order, using whatever values of other states happen to be available. The values of some states may be backed up several times before the values of others are backed up once. To converge correctly, an asynchronous algorithm must continue to backup the values of all the states. These algorithms allow greater flexibility in selecting states to which backup operations are applied.

One version of asynchronous value iteration backs up the value, in place, of only one state s_k on each step k, using the value iteration backup

$$V_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\, V_k(s')],$$

as seen in the previous section. If $0 \leq \gamma \leq 1$, asymptotic convergence to V^* is guaranteed given only that all states occur an infinite number of times.

Avoiding sweeps does not necessarily mean getting away with less computation. It just means that an algorithm does not need to get stuck in any long sweep before it can make progress improving a policy. Some states may not need their values backed up as often as others. It might even try to skip backing up some states entirely if they are not relevant to optimal behaviour.

Asynchronous algorithms make it easier to blend computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm at the same time that the agent is experiencing the MDP by using the agent's experience to determine the states which the DP algorithm backs up (Sutton and Barto, 1998*a*). At the same time the latest value and policy information from the DP algorithm can guide the agent's decision making.

3.8 Generalised policy iteration

Generalised policy iteration (GPI) is used to refer to the notion of interacting policy evaluation and policy improvement processes revolving around an approximate policy and an approximate value function. Almost all RL methods are well described as GPI. They all have distinctive policies that are always being improved with respect to the value function and the value function always being driven towards the value function for the policy.

GPI consists of two interacting processes executing at the same time. One making the value function consistent with the current policy (policy evaluation), the other making the policy greedy with respect to the current value function (policy improvement). These two processes alternate, each completing before the other begins, but it is not really necessary to wait for one to finish. As long as both processes continue to update all states, the ultimate result is convergence to the optimal value function and an optimal policy.

If both the evaluation process and the improvement process no longer produce any changes, it means that there are no better evaluations or improvements left to be made. Then the value function and policy must be optimal. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function and performs some form of policy improvement, changing the policy to make it better. Both processes stabilise only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation holds and that the policy and the value function are optimal.

With GPI we have another case where we need to find a balance between two processes, just as with the trade off between exploration and exploitation. We cannot only use the evaluation process or only the improvement process, because these two processes pull in

opposite directions. Making the policy greedy with respect to the value function makes the value function incorrect for the changed policy, but making the value function consistent with the policy causes the policy to no longer be greedy. These two processes interact to find a single joint solution: the optimal value function and the optimal policy.

The interaction between the evaluation and improvement processes in GPI can be thought of in terms of two constraints or goals (two lines in 2D space):

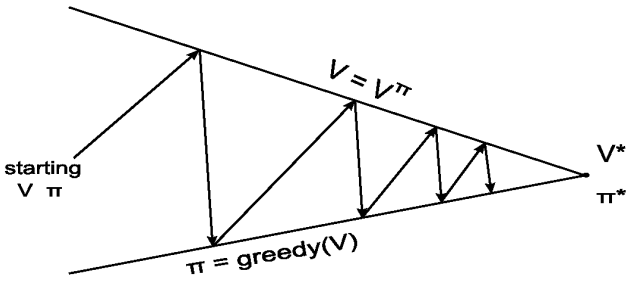


Figure 3.4: Interaction between evaluation and improvement processes

Each process drives the value function or policy towards one of the lines representing a solution to one of the two goals. The two lines cross, illustrating the dependencies between the goals. Driving directly towards one goal causes some movement away from the other goal. The two processes together achieve the overall goal of optimality even though neither attempts to achieve it directly.

3.9 How efficient is DP?

Classical DP algorithms are not used frequently because they assume a perfect model, as well as being computationally expensive. They are also known to converge slower than modern DP methods (Dai and Goldsmith, 2009) and this could be an issue for problems with large state sets.

Another cause for concern, especially in very large problems, is the *curse of dimensionality*. The curse of dimensionality refers to the tendency of a state space to grow exponentially in its dimension, in this case, in the number of state variables (Sutton, 2004). This is a problem for methods such as DP and other table-based RL methods whose complexity scales linearly with the number of states. DP is, however, still better suited to handling large state spaces than methods such as direct search and linear programming, because DP requires less memory than these two methods.

In practice, both policy iteration and value iteration are widely used and it is not clear which is better overall. These methods usually converge much faster than their theoretical worst-

case run times, especially if they are started with good initial value functions or policies (Sutton and Barto, 1998*a*).

Asynchronous DP methods are often preferred for problems with large state spaces, as completing even a single sweep of a synchronous method requires computation and memory for every state. Asynchronous methods and other variations of GPI may find good or optimal policies much faster than synchronous methods can.

3.10 Concluding remarks

In this chapter we looked at the general theory behind DP including policy evaluation, policy improvement, policy iteration, value iteration, asynchronous DP, and GPI. We also considered the efficiency of DP methods.

Policy evaluation refers to the iterative computation of the value functions for a given policy, while *policy improvement* refers to the computation of an improved policy given the value function for that policy. By combining these two computations, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing a full backup operation on each state. Each backup updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Full backups are like Bellman optimality equations turned into assignment statements. Just as there are four primary value functions (V^π, V^*, Q^π, Q^*), there are four corresponding Bellman equations and four corresponding full backups.

All DP methods update estimates on the basis of other estimates, and this is called *bootstrapping*. Many RL methods perform bootstrapping, even those that do not require (as DP does) a complete and accurate model of the environment.

Chapter 4

Monte Carlo methods

4.1 Introduction

In this chapter we discuss the second family of methods for solving the Reinforcement Learning (RL) problem, namely Monte Carlo (MC) methods. We look at the theory behind MC methods and at policy evaluation and the estimation of action values. We also look at on-policy and off-policy control and then illustrate some of the algorithms with examples. We end the chapter with a short summary of what was discussed.

4.2 Theory of MC methods

MC methods are different from the Dynamic Programming (DP) methods discussed in Chapter 3, mainly because they do not assume a complete knowledge of the environment. The only requirement is experience; sample sequences or simulated interactions with an environment. To learn from on-line experience, we do not need prior knowledge of the environment's dynamics, but can still achieve optimal behaviour. We do not need the complete probability distributions of all possible transitions as required by DP methods (Sutton and Barto, 1998*a*).

MC methods solve the RL problem based on averaging sample returns. To ensure that well-defined returns are available, we define MC methods only for episodic tasks. We assume that the experience is divided into episodes and that all episodes eventually terminate. It is only at the end of an episode that value estimates and policies are changed. MC methods are incremental in an episode-by-episode sense, but not in a step-by-step sense.

Despite the differences between MC and DP methods, the most important ideas carry over from DP to the MC case.

4.3 Evaluating the policy

We start by considering MC methods for learning the state-value function for a given policy. As we know by now, the value of a state is the expected return starting from that state. As more returns are observed, the average should converge to the expected value.

Suppose we have a set of episodes that were obtained by following policy π and passing through state s. We now want to estimate the value $V^\pi(s)$ of this state s under the policy π. Each occurrence of state s in an episode is called a *visit* to s. The *every-visit* MC method estimates $V^\pi(s)$ as the average of the returns following all the visits to s in a set of episodes. Within a given episode, the first time s is visited is called the *first visit* to s. The *first-visit* MC method averages the returns following only first visits to s. This method will be the focus of the chapter.

The first-visit MC method converges to $V^\pi(s)$ as the number of first visits to s goes to infinity. The sequence of averages of these estimates converge to their expected value. Algorithm 4 shows the corresponding algorithm.

Algorithm 4 First-visit MC method for estimating V^π

```
Initialise:
π ← policy to be evaluated
V ← arbitrary state-value function
Returns(s) ← empty list, for all s ∈ S

loop
  Generate an episode using π

  for each state s appearing in the episode do
    R ← return following the first occurrence of s
    Append R to Returns(s)
    V(s) ← average(Returns(s))
  end for
end loop
```

4.3.1 Blackjack example

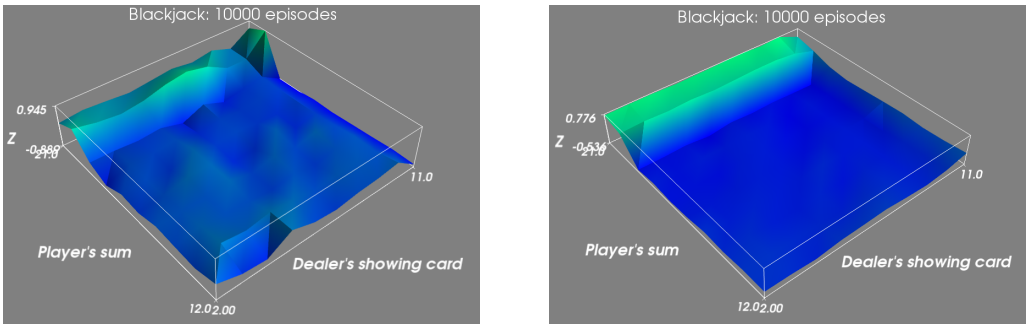
The first example we are looking at in this chapter is the simplified version of the game of blackjack. In this particular example, the player competes independently against the dealer. The objective of the game is to obtain cards the sum of whose numerical value is as close to 21 as possible without exceeding 21. All face cards count as 10, and the ace can count either as one or as 11. If the player holds an ace that he may count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11, because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made, because obviously the player should always hit (Sutton and Barto, 1998*a*).

The game starts with two cards dealt to the player and two cards dealt to the dealer. Both of the player's cards are face up, but only one of the dealer's cards is face up. If the player has 21 immediately, it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, he can request additional cards, one by one (*hits*), until he either stops (*sticks*) or exceeds 21 (*goes bust*). If he goes bust, he loses. If he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or higher, and hits otherwise. If the dealer goes bust, the player wins. Otherwise the outcome– win, lose or draw, is determined by whose final sum is closer to 21.

Playing blackjack is formulated as an episodic finite Markov decision process (MDP). Each game of blackjack is an episode and rewards are +1, -1 and 0 for winning, losing and drawing, respectively. All the rewards within a game are zero, and we do not discount ($\gamma = 1$), therefore the terminal rewards are also the returns. The player's actions are $a \in \{\texttt{hit}, \texttt{stick}\}$. The states depend on the player's cards and the dealer's showing card. We assume that the cards are dealt from an infinite deck so that there is no advantage to keeping track of the cards already dealt. The player makes decisions on the basis of three variables: his current sum (12-21), whether or not he holds a usable ace, and the dealer's one showing card (2-11). This makes for a total of 200 states.

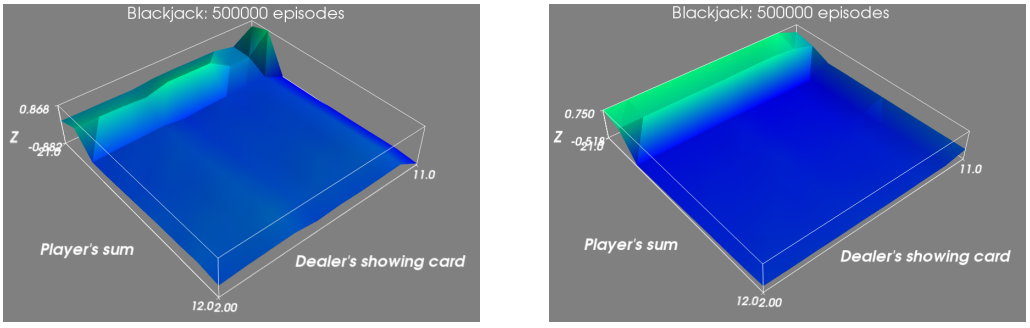
Consider for a moment a starting sum of 11. If the player decides to hit, the highest value he can possibly draw is a 10 (an ace would yield a sum greater than 21, thus we count it as one). If the player draws this maximum value card, his sum is 21 and he sticks. Now consider the case where he has a starting sum of 12. If the player draws a 10 he will go bust, so there is a chance that the player will go bust if he has a sum of 12, whereas with a sum of 11 there was no such chance. This means that with a starting sum of less than 12, there is no decision to be made: the player always hits. With a sum of at least 12 the player needs to make a decision whether he hits or sticks.

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a MC approach, we simulate many blackjack games using the policy and average the returns following each state. In this particular problem the same state never occurs more than once within one episode, so there is no difference between the first-visit MC method and the every-visit MC method. We obtain the estimates for the value function in Figure (4.1) after 10,000 games. Figure (4.2) shows the results for 500,000 games. The estimates for states with a usable ace are less certain and less regular because these states are less common. This causes the less than smooth effect in both Figure (4.1)(a) and Figure (4.2)(a). After 500,000 games the value function is well approximated and the graph much smoother.



(a) Usable ace (b) No usable ace

Figure 4.1: Approximate state-value functions for blackjack after 10,000 episodes



(a) Usable ace (b) No usable ace

Figure 4.2: Approximate state-value functions for blackjack after 500,000 episodes

From Figure (4.1)(a) we see that when the player's sum is low (about 12 or 13) and the dealer's showing card is low (about 2-5), the player has a good chance of losing. The reason for this is that the player has a better chance of going bust when starting out with a sum of 12 or 13, because he has to hit until he reaches at least 20. The same is true when the player's sum is high (about 19) and the dealer's showing card is very low (about 2) or mid-range (about 6). When the dealer has a usable ace, the chances of the player winning are very slim.

The player has a very good chance of winning when his sum is 20 or 21 (when he sticks according to the original policy) and the dealer's showing card is 11. The reason for this is, the dealer always sticks on 17 or higher, so if he draws anything higher than a 6 (60% of the cards), his total will be 17 (or greater) and he will have to stick, thereby losing to the player's 20 or 21. From Figure (4.1)(b) we can see that the only way the player can win for certain, given the original policy, is to have a sum of 20 or higher.

From Figure (4.2)(a) we see that the player will win the most blackjack games when he has 20 or 21 and the dealer has a usable ace. Figure (4.2)(b) is very similar to Figure (4.1)(b) and the same result follows.

Although we have complete knowledge of the environment in this task, it is not easy to apply DP policy evaluation to compute the value function, whereas generating the sample games required by MC is easy.

4.4 Estimating the action values

When there is a model of the environment available, state values alone are enough to determine a policy. However, when a model of the environment is not available, these state values are not enough. In this case we rather estimate the action values. We only need to look ahead one step and choose the action that leads to the highest return. One of our primary goals is thus to estimate Q^*.

The policy evaluation problem for action values is to estimate $Q^\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π. The first-visit MC method averages the returns following the first time in each episode that the state s was visited and the action a was selected. The method converges to the true expected values as the number of visits to each state-action pair approaches infinity.

A problem with this is that many relevant state-action pairs may never be visited. When following π we observe returns only for one of the actions from each state. With no returns to average, the MC estimates of the other actions will not improve with experience. To compare alternatives we need to estimate the value of all the actions from each state, not just the one we currently favour.

This is the general problem of maintaining exploration. For policy evaluation to work for action values, we must assure continual exploration. To do this, we specify that each episode start with a state-action pair and that every such pair has a non-zero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times. We call this the *assumption of exploring starts* (Sutton and Barto, 1998*a*).

The assumption of exploring starts cannot be relied upon in general, particularly when learning directly from real interactions with an environment. The most common alternative to assuring that all state-action pairs are encountered is to consider only policies that are stochastic with a non-zero probability of selecting all actions.

4.5 MC control

We proceed according to the same pattern as in the DP chapter when using MC estimation to approximate optimal policies. Thinking back to generalised policy iteration (GPI), we maintain both an approximate policy and an approximate value function. The value function is continuously altered to more closely approximate the value function for the current policy and the policy is repeatedly improved with respect to the current value function.

An MC version of policy iteration would be to perform alternating complete steps of policy evaluation and policy improvement. We begin with an arbitrary policy π and end with the optimal policy π^* and optimal action-value function Q^*. The policy evaluation is done exactly as in the previous section. The approximate action-value function converges to the true function when many episodes are experienced.

Policy improvement is done by making the policy greedy with respect to the current value function. Remember that in Section 4.4 we said that if there is no model, the action-value function should rather be computed. In this case we have an action-value function, and therefore no model is needed to construct the greedy policy. For any action-value function Q, the corresponding greedy policy is the one that, for each state $s \in \mathcal{S}$, deterministically chooses an action with maximal Q-value:

$$\pi(s) = \arg \max_a Q(s, a).$$

Policy improvement can then be done by constructing each π_{k+1} as the greedy policy with respect to Q^{π_k}. The policy improvement theorem then applies to π_k and π_{k+1}, because, for all states $s \in \mathcal{S}$,

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \arg \max_a Q^{\pi_k}(s, a)) \\ &= \max_a Q^{\pi_k}(s, a) \\ &\geq Q^{\pi_k}(s, \pi_k(s)) \\ &= V^{\pi_k}(s). \end{aligned}$$

The theorem assures us that each π_{k+1} is uniformly better than π_k, unless it is equal to π_k, in which case they are both optimal policies. This then assures us that the overall process converges to an optimal policy and the optimal value function.

We made two unlikely assumptions in order to easily obtain this guarantee of convergence for the MC method. One was that the episodes have exploring starts. The other was that policy evaluation could be done with an infinite number of episodes. To obtain a practical algorithm we have to remove both assumptions.

Focusing on the assumption that policy evaluation works well on an infinite number of episodes, there are two ways to remove this assumption: the one is to approximate Q^{π_k} in each policy evaluation and the other one to forgo trying to complete policy evaluation before returning to policy improvement. Bounds are obtained on the magnitude and probability of error in the estimates, and then a number of steps are taken during each policy evaluation to assure that these bounds are sufficiently small. This guarantees correct convergence up to some level of approximation.

The second approach moves the value function toward Q^{π_k} on each evaluation step, but we do not expect to actually get close except over many steps. One extreme form of the

idea is value iteration, in which only one iteration of iterative policy evaluation is performed between each step of policy improvement.

For MC policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. Algorithm 5 gives an outline of the MC control algorithm assuming exploring starts.

Algorithm 5 MCES (assuming exploring starts)

Initialise, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list

loop
  Generate an episode using exploring starts and π
  for each pair s, a appearing in the episode **do**
    $R \leftarrow$ return following the first occurrence of s, a
    Append R to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  end for

  for each s in the episode **do**
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$
  end for
end loop

In MCES, all the returns for each state-action pair are accumulated and averaged, no matter which policy was in use when they were observed. MCES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal.

4.5.1 Solving blackjack

Going back to our blackjack example, we know the episodes are all simulated games. It is easy to ensure exploring starts that include all possibilities. We use as initial policy the one where the player sticks only on 20 or 21. The initial action-value function can be zero for all state-action pairs. In Figures (4.3) and (4.4) the optimal state-value functions are shown. Again we see that the graph for the case of a usable ace is much less smooth than that of the case with a non-usable ace. We can see that, in comparison to the previously generated blackjack figures, these graphs are much smoother with less jagged lines.

Figure (4.3) shows us that if the player's sum is mid-range (12-16), and the dealer's showing card is either mid-range or 11, then the player has a big chance of losing. The reason for

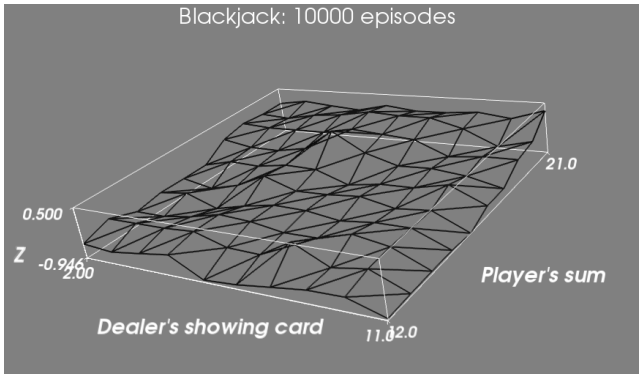


Figure 4.3: Optimal state-value functions for blackjack after 10,000 episodes with a usable ace

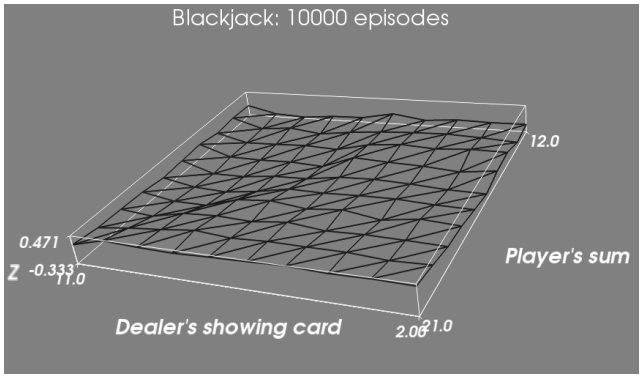


Figure 4.4: Optimal state-value functions for blackjack after 10,000 episodes with no usable ace

this is that the player might go on hitting, thus going bust. If, on the other hand, the player has a mid-range to high sum (16 and greater) and the dealer's showing card is mid-range (6-9), then the player has a good chance of winning. The player's best chance of winning appears to be when he has 20 or 21 and the dealer has a mid-range card. The reason for this might again be due to the dealer's policy of sticking on 17 and higher.

Looking at Figure (4.4), it seems to be that the higher the dealer's showing card, the higher the player's sum has to be in order for him to win.

4.6 On-policy MC control

Exploring starts is an unlikely assumption that we wish to avoid. The only way that we can be sure that all actions are sampled infinitely often, is for the agent to select them all. Two approaches ensure this: on-policy and off-policy methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions. Here we look at these

on-policy methods. We look at off-policy methods in Section 4.8.

In on-policy control methods the policy is usually soft, meaning that $\pi(s,a) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$. One variation of on-policy methods is to gradually shift the policy toward a deterministic optimal policy. The on-policy method presented here uses ϵ-greedy policies. Most of the time an ϵ-greedy policy chooses a greedy action (an action that has a maximal estimated action value), but with probability ϵ it selects an action at random. If $\epsilon = 0.1$ then the policy chooses a random action 10% of the time, thus exploring. All non-greedy actions are given the minimal probability of being selected, $\frac{\epsilon}{|\mathcal{A}|}$, and the remaining bulk of the probability, $1-\epsilon+\frac{\epsilon}{|\mathcal{A}|}$, is given to the greedy actions. The ϵ-greedy policies are examples of ϵ-soft policies, defined as policies for which $\pi(s,a) \leq \frac{\epsilon}{|\mathcal{A}|}$ for all states and actions, for some $\epsilon > 0$.

The main idea of on-policy MC control is still that of GPI. First-visit MC methods are used to estimate the action-value function Q for the current policy π. Without the assumption of exploring starts we cannot improve the policy by making it greedy with respect to the current value function, because that prevents further exploration of non-greedy actions. GPI requires that the policy be moved towards, and not taken all the way to a greedy policy. Here we move it only to an ϵ-greedy policy. For any ϵ-soft policy π, any ϵ-greedy policy with respect to Q^π is guaranteed to be better than or equal to π. The policy improvement theorem assures us of the improvement from any ϵ-greedy policy with respect to Q^π over any ϵ-soft policy π. Let π' be the ϵ-greedy policy. From Chapter 3 it follows that $Q^\pi(s,\pi'(s)) = V^\pi$. Thus by the policy improvement theorem, $\pi' \geq \pi$. Equality holds only when both π' and π are optimal among the ϵ-soft policies. The reader is referred to Appendix B.1 for the derivation of this result.

From (Sutton and Barto, 1998*a*) it can be shown that policy iteration works for ϵ-soft policies however we will not go any further into that. Using the natural notion for ϵ-soft policies, we are assured of improvement on every step, except when the best policy has been found. The analysis is independent of how the action-value functions are determined at each stage, but it does assume that they are computed exactly. We only achieve the best policy among the ϵ-soft policies, but we eliminated the assumption of exploring starts. Algorithm 6 gives a complete ϵ-soft on-policy MC control algorithm.

Algorithm 6 ϵ-soft on-policy MC control

Initialise, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$\pi \leftarrow$ an arbitrary ϵ-soft policy
$Returns(s,a) \leftarrow$ empty list

loop
  Generate an episode using π
  for each pair s, a appearing in the episode **do**
    $R \leftarrow$ return following the first occurrence of s, a
    Append R to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  end for

  for each s in the episode **do**
    $a^* \leftarrow \arg\max_a Q(s,a)$
    for all $a \in \mathcal{A}(s)$ **do**

$$\pi(s,a) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)|, & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

    end for
  end for
end loop

4.7 Evaluating one policy while following another

Suppose we wish to estimate the value function V^π or Q^π for a policy π, but all we have are episodes generated by a different policy π', where $\pi' \neq \pi$. Can we learn the value function for a policy given only experience "off" the policy?

In order to use episodes from π' to estimate values for π, we require that every action taken under π is also taken under π'. It is thus required that $\pi(s,a) > 0$ implies $\pi'(s,a) > 0$. In the episodes generated using π', consider the i^{th} first visit to state s and the complete sequence of states and actions following that visit. Let $p_i(s)$ and $p'_i(s)$ denote the probabilities of that complete sequence happening, given policies π and π' and starting from s. Let $R_i(s)$ denote the corresponding observed return from state s.

To average these returns to obtain an unbiased estimate of $V^\pi(s)$, each return need only be weighted by its relative probability of occurring under π and π', that is, by $\frac{p_i(s)}{p'_i(s)}$. The desired MC estimate after observing n_s returns from state s is then:

$$V(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)}}. \tag{4.1}$$

The probabilities $p_i(s)$ and $p'_i(s)$ are normally unknown in applications of MC methods. Only

their ratio is needed, and that can be determined with no knowledge of the environment's dynamics. Let $T_i(s)$ be the time of termination of the i^{th} episode involving state s. Then

$$p_i(s_t) = \prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k)\mathcal{P}^{a_k}_{s_k s_{k+1}}$$

and

$$\frac{p_i(s_t)}{p'_i(s_t)} = \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k, a_k)}{\pi'(s_k, a_k)}.$$

Thus the weight needed in equation (4.1) depends only on the two policies and not at all on the environment's dynamics.

4.8 Off-policy MC control

What sets on-policy methods apart from off-policy methods is that on-policy methods estimate the value of a policy while using that policy for control. In off-policy methods these two functions are separated. The policy used to generate behaviour (*behaviour policy*) may be unrelated to the policy that is evaluated and improved (*estimation policy*). The advantage of this separation is that the estimation policy may be deterministic (*e.g.* greedy), while the behaviour policy can continue to sample all possible actions.

Off-policy control methods estimate the value function for one policy while following another, just as described in Section 4.7. It follows the behaviour policy while learning about and improving the estimation policy. The behaviour policy must have a non-zero probability of selecting all actions that might be selected by the estimation policy. To explore all possibilities, the behaviour policy must be soft.

Algorithm 7 gives a complete off-policy MC control algorithm where Q^* is computed based on GPI. The behaviour policy is maintained as an arbitrary soft policy, while the estimation policy π is the greedy policy with respect to Q, an estimate of Q^π.

The potential problem is that the method learns only from the back ends of episodes, after the last non-greedy action. Learning will be slow if non-greedy actions are frequent, particularly for states appearing in the early parts of long episodes.

Algorithm 7 Off-policy MC control

Initialise, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$N(s,a) \leftarrow 0$; Numerator and
$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$
$\pi(s) \leftarrow$ arbitrary deterministic policy

loop
  Select a policy π' and use it to generate an episode:
    $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
  $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
  for each pair s, a appearing in the episode at time later than τ **do**
    $t \leftarrow$ the time of first occurrence of s, a such that $t \geq \tau$
    $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
    $N(s,a) \leftarrow N(s,a) + wR_t$
    $D(s,a) \leftarrow D(s,a) + w$
    $Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
  end for

  for each $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$
  end for
end loop

4.9 Concluding remarks

In this chapter we reviewed several methods for solving the RL problem with MC methods. MC methods are another family of methods for estimating value functions and discovering optimal policies.

As MC operates on sample experiences, it can be used for direct learning without a model. MC methods learn value functions and optimal policies from experience in the form of sample episodes which holds at least three advantages over DP methods. MC methods can be used to learn optimal behaviour directly from interaction with the environment, with no model of the environment's dynamics. The second advantage is that it can be used with simulation or sample models. It is easy to simulate sample episodes for many applications even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. The third advantage is that MC methods are easy and efficient to focus on a small subset of the states. A specific subset can be evaluated without going to the trouble of accurately evaluating the rest of the state set.

A fourth possible advantage is that MC methods may be less harmed by violations of the Markov property. This is because they do not bootstrap– they do not update their value estimations on the basis of the value estimates of successor states.

We follow the overall scheme of interacting processes of policy evaluation and policy im-

provement (GPI) when designing MC control methods. MC methods provide an alternative policy evaluation process. Rather than using a model to compute the value of each state, they average many returns that start in that state. The state's value is the expected return and so this average can become a good approximation to the value. In control methods we are particularly interested in approximating action-value functions, because these can be used to improve the policy without having a model of the environment's dynamics. MC mixes policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Continual exploration remains a key issue. We cannot continuously select actions that are currently the best. If we do that, no new returns will be obtained for alternative actions and we may never find out that they are actually better. We could ignore this problem by assuming that episodes start with state-action pairs randomly selected to cover all possibilities. Two better approaches are on-line policy and off-line policy methods. On-line policy methods learn a policy while following it. The agent always explores and tries to find the best policy that still explores. Off-policy methods basically follow one policy while evaluating another. The agent also explores, but learns an optimal policy while following another, mostly unrelated, policy.

Chapter 5

Temporal-difference learning

5.1 Introduction

Temporal-Difference, or TD, learning is a third family of popular methods for solving the Reinforcement Learning (RL) problem. We divide the overall problem into a prediction problem and a control problem and show how TD learning methods can be used as alternatives to Monte Carlo (MC) methods for solving the prediction, as well as the control, problem. The prediction problem refers to policy evaluation; predicting how good a policy is based on the value function, and the control problem refers to improving the policy so as to find an optimal policy.

5.2 Theory of TD learning methods

The goal of learning is to generate the optimal actions leading to maximal returns (Tesauro, 1995). Returns are, per definition, given at the end of an episode. The question we ask is, can we update the expected values earlier than at the end of the episode?

A class of methods for solving this is called *Temporal-Difference* (TD) learning methods. The basic idea of TD methods is that the learning is based on the difference between successive predictions in time. In other words, the learner's current prediction is updated by reward at the next time step. An error is defined at every time step, based on the difference between two successive predictions. This error drives the learning. Suppose we detect a prediction error at a given time step. Then there is an exponentially decaying feedback of the error backwards in time so that previous estimates for previous states are also corrected.

TD learning is a combination of MC ideas and Dynamic Programming (DP) ideas (Sutton and Barto, 1998*a*). TD methods are like MC methods in the sense that they can learn directly from raw "events" without a model of the environment's dynamics. They are also

like DP methods in the sense that they bootstrap. Estimates are updated in part on other learned estimates, without waiting for a final outcome. These ideas and methods blend into each other and can be combined in many ways.

While TD is designed to deal with prediction problems, it is also used to solve optimal control problems (Woergoetter and Porr, 2008). Thinking back to the start of the chapter, the reader is reminded that "control" refers to policy improvement; finding an optimal policy. TD learning converges to the value function V^* if all states are visited "often enough". The advantage of TD learning is that the methods do not require a model of the environment, only experience, as with MC (Sutton and Barto, 1998*b*). Unlike MC, TD methods can be fully incremental. We can learn before knowing the final outcome which uses less memory and less computation.

We start by focusing on policy evaluation (the prediction problem); that of estimating the value function V^π for a given policy π. For the control problem, DP, TD and MC methods all use some variation of generalized policy iteration (GPI) (Sutton and Barto, 1998*a*). The differences in the methods are mainly due to their approaches to the prediction problem.

5.3 Evaluating the policy

Assume that we have an estimate $V(s)$ of $V^\pi(s)$ for all states s. The problem is one of updating $V(s_t)$, given that we are in state s_t at time t. Since $V^\pi(s_t)$ is the expected value of the return R_t after visiting s_t, we either need to calculate, or estimate, R_t in order to find an approximation of $E\{R_t\}$. A simple every-visit MC method called *constant-α MC*, is:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)], \tag{5.1}$$

where R_t is the actual return following time t and α is a constant step-size parameter (Sutton and Barto, 1998*a*). MC waits until the return at the end of the visit is known, then uses a point wise estimate for $V(s_t)$ as in equation (5.1). DP uses bootstrapping as in equation (3.3). Note that with DP the expected value is available since we have a model.

Where MC methods must wait until the end of the episode to determine the change to $V(s_t)$, TD methods only wait until the next time step. At time $t+1$, TD methods immediately form a target and make an update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, TD(0), is

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]. \tag{5.2}$$

Comparing equations (5.1) and (5.2), we see that the target for the MC update is R_t, but the target for the TD update is $r_{t+1} + \gamma V_t(s_{t+1})$.

As previously stated, TD is a bootstrapping method because it bases its update partly on

an existing estimate, just like DP. We know that

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} \tag{5.3}$$
$$= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s\}. \tag{5.4}$$

As seen in equation (5.1), MC methods use an estimate of equation (5.3) as a target, while DP methods use an estimate of equation (5.4) as a target. We say the MC target is an estimate, because the expected value in equation (5.3) is unknown. The sample return needs to be used to estimate the expected value, and not the real expected return. The DP target is also an estimate, not because the expected value is unknown, but because $V^\pi(s_{t+1})$ is unknown and we use the current estimate $V_t(s_{t+1})$ instead. The expected value is provided by a model of the environment. The TD target is an estimate for both reasons. It samples the expected value in equation (5.4) and uses the current estimate V_t instead of the true V^π. TD combines the sampling of MC with the bootstrapping of DP. Algorithm 8 gives an outline of tabular TD(0) for estimating V^π.

Algorithm 8 Tabular TD(0) for estimating V^π

Initialize $V(s)$ arbitrarily, π the policy to be evaluated
repeat
  (for each episode)
  Initialize s
  repeat
    (for each step of episode)
    $a \leftarrow$ action given by π for s
    Take action a; observe reward r, and next state s'
    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
    $s \leftarrow s'$
  until end of episode
until s is terminal

TD updates and MC updates are *sample backups*. Sample backups look ahead to a sample successor state, using the value of the successor and the reward along the way to compute a backed-up value, and then change the value of the original state accordingly. Sample backups are based on a single sample successor rather than on a complete distribution of all possible successors as with DP's full backups.

5.3.1 TD(0) Example

We illustrate Algorithm 8 with the following simple example. Suppose we have six states $s_0, s_1, s_2, s_3, s_4, s_5$, where s_0 is a terminal state. Further suppose that we have two actions a_1 and a_2, where a_1 means that we are moving left and a_2 that we are moving right. Figure (6.1) illustrates these states and actions. Note that the shaded cells on either side are terminal, and can be thought of as the same state, s_0.

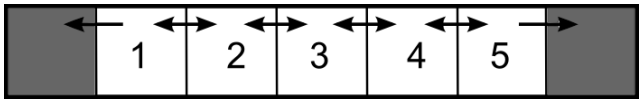


Figure 5.1: A simple walk

If we reach the terminal state, a reward of +1 is awarded, otherwise the reward is 0. We are working with an undiscounted ($\gamma = 1$) episodic task. An episode ends when the terminal state is reached.

Applying Algorithm 8 to this example, we estimate V^π, that is, we want to know what the value of our states are when following policy π. What distinguishes TD from Monte Carlo is the fact that it does not wait until the episode is finished to update the value function, it only waits until the next time step.

Implementing the prediction example

We initialise $V(s)$ as zero for all states $s \in \mathcal{S}$ and choose the policy π to be evaluated as the one that tells us to move left at every state, $\pi = \{1, 1, 1, 1, 1\}$. For the first episode we initialise the state as $s = s_1$ and take action $a = a_1$. We observe a reward of $r = +1$ and the next state $s' = s_0$. The value estimate $V(s_1)$ is updated to $V(s_1) = \alpha$, where α is a constant stepsize parameter between zero and one. The next state to be evaluated is $s = s_0$, but s_0 is our terminal state, therefore we end this episode and start the next one.

The second episode starts with $s = s_2$. We take action $a = a_1$, observe the reward to be $r = 0$, and the next state $s' = s_1$. The value estimate for $V(s_2)$ is α^2 and the next state is set to $s = s_1$. We again take action $a = a_1$ and observe the reward to be $r = +1$ and the next state $s' = s_0$. The value estimate for $V(s_1)$ was updated to $3\alpha - 3\alpha^2 + \alpha^3$ and the next state to be evaluated is set to $s = s_0$. But again, s_0 is our terminal state, thus ending episode two.

We continued in this way for states $s = s_3, s_4$ and s_5. Table 5-I shows the value estimates updated in each episode.

$V(s_i)$	Episode 1	Episode 2	Episode 3	Episode 4	Episode 5
$V(s_1)$	α	$\alpha(2-\alpha)$	$\alpha(3-3\alpha+\alpha^2)$	$\alpha(4-6\alpha+4\alpha^2-\alpha^3)$	$\alpha(5-10\alpha+10\alpha^2-5\alpha^3+\alpha^4)$
$V(s_2)$	0	α^2	$\alpha^2(3-2\alpha)$	$\alpha^2(6-8\alpha+3\alpha^2)$	$\alpha^2(10-20\alpha+15\alpha^2-4\alpha^3)$
$V(s_3)$	0	0	α^3	$\alpha^3(4-3\alpha)$	$\alpha^3(10-15\alpha+6\alpha^2)$
$V(s_4)$	0	0	0	α^4	$\alpha^4(4-3\alpha)$
$V(s_5)$	0	0	0	0	α^5

Table 5-I: TD(0): Estimated value function after one run in terms of α

The first time a state's value is changed, its value estimate is increased by a power of α. The reason for this is that the value estimates are initially zero and the only non-zero term in the update equation is the term concerned with the next state's value estimate, $V(s')$. For

example, $V(s_1)$ started out as zero and was then updated to α. This update was used to calculate $V(s_2)$, thus obtaining α^2. The update for $V(s_3)$ was calculated using $V(s_2) = \alpha^2$, thus obtaining α^3 and so forth.

Table 5-II shows these value estimates for the case where $\alpha = 0.1$.

$V(s_i)$	Episode 1	Episode 2	Episode 3	Episode 4	Episode 5
$V(s_1)$	0.1000	0.1900	0.2710	0.3439	0.4095
$V(s_2)$	0	0.0100	0.0280	0.0523	0.0815
$V(s_3)$	0	0	0.0010	0.0037	0.0086
$V(s_4)$	0	0	0	0.0001	0.0004
$V(s_5)$	0	0	0	0	0.00001

Table 5-II: TD(0): Estimated value function after one run

5.4 Advantages of evaluating the policy with TD

Although TD methods bootstrap like DP, they have an advantage over DP in that they do not require a model of the environment, nor of its reward and transition probabilities.

TD methods' advantage over MC is that they are naturally implemented in an on-line, fully incremental way. With MC you must wait until the end of an episode, because only then the return is known, but with TD you only need to wait one time step. Some applications have very long episodes, and delaying all learning until an episode is finished can take very long. Other applications are continuing tasks and have no episodes at all. Some MC methods must ignore or discount episodes on which experimental actions are taken and this can greatly slow down learning. TD methods are much less vulnerable to these problems because they learn from each transition regardless of what subsequent actions are taken.

5.5 Optimality of TD(0)

Suppose we only have a finite amount of experience available, say 20 episodes or 200 time steps. A common approach with incremental learning methods such as TD methods, is to present the experience repeatedly until the method converges. Given an approximate value function V, the increments specified by

$$V(s_t) = V(s_t) + \alpha[R_t - V(s_t)] \tag{5.5}$$

or

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{5.6}$$

where equation (5.5) is the constant-α MC method and equation (5.6) is the TD(0) method, are computed for every time step t at which a non-terminal state is visited. The value

function is changed only once, after which all the available experience is processed again with the new value function to produce a new increment. This is done until the value function converges. This is called *batch updating* because updates are made only after processing each complete batch of training data.

Under batch updating, TD(0) converges deterministically to an answer independent of the step-size parameter α, as long as α is chosen to be sufficiently small. The constant-α MC method also converges deterministically under the same conditions, but to a different answer. Under batch training, constant-α MC methods converge to values $V(s)$, that are sample averages of the actual returns experienced after visiting each state s. These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set.

Batch MC methods always find the estimates that minimize mean-squared error on the training set, where batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. The maximum-likelihood estimate of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed from the observed episodes. The estimated transition probability from i to j is the fraction of observed transitions from i that went to j. The associated expected reward is the average of the rewards observed on those transitions.

Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate*. It is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. Batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than MC methods. In batch form, TD(0) is faster than MC methods because it computes the true certainty-equivalence estimate. Non-batch TD(0) may also be faster than constant-α MC because it is moving toward a better estimate, even though it is not getting all the way there.

Although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute directly. If N is the number of states, then just forming the maximum-likelihood estimate of the process may require N^2 memory. Computing the corresponding value function requires on the order of N^3 computational steps if done conventionally. TD methods can approximate the same solution using memory of no more than N and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

5.6 Sarsa: On-policy TD control

In previous sections we estimated the value function, $V^\pi(s)$ (the prediction problem). Now we move on to find an optimal policy (the control problem). As before we use GPI. Again we are faced with the need to trade off exploration against exploitation. The approaches fall into two categories, namely *on-policy* and *off-policy* control. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. For that we need $Q^\pi(s,a)$ for all states s and all actions a, instead of $V^\pi(s)$. This can be done by using basically the same TD method described for estimating V^π. We do not consider state to state transitions any more, we consider transitions from state-action pairs to state-action pairs and learn the value of these state-action pairs. Theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]. \tag{5.7}$$

The update is done after every transition from a non-terminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1},a_{t+1})$ is defined as zero. This update rule uses every element of the events, $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$, that make up a transition from one state-action pair to the next. The quintuple gives rise to the name *Sarsa*.

We continually estimate Q^π for the behaviour policy π, and at the same time change π toward greediness with respect to Q^π. Algorithm 9 gives the general form of the Sarsa control algorithm. We use ϵ-greedy policies for which $\pi(s,a) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$, where ϵ is the greediness parameter, $\pi(s,a)$ is the probability of taking action a when in state s under a policy π, and $\mathcal{A}(s)$ is the action space.

Algorithm 9 Sarsa: On-policy TD control algorithm

```
Initialize Q(s,a) arbitrarily
repeat
  (for each episode)
  Initialize s
  Choose a from s using policy derived from Q (e.g. ε-greedy)
  repeat
    (for each step of episode)
    Take action a; observe reward r, and next state s'
    Choose a' from s' using policy derived from Q (e.g. ε-greedy)
    Q(s,a) ← Q(s,a) + α[r + γQ(s',a') − Q(s,a)]
    s ← s'; a ← a'
  until end of episode
until s is terminal
```

We could use ϵ-greedy or ϵ-soft policies. Sarsa converges with probability one to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number

of times and the policy converges in the limit to the greedy policy. This can be arranged, for example, with ϵ-greedy policies by setting $\epsilon = \frac{1}{t}$ (Sutton and Barto, 1998*a*).

Sarsa does not have the problem of finding a policy that causes the agent to stay in the same state and being in a never ending episode, like MC. Sarsa quickly learns during the episode that such policies are poor and then switch to something else.

5.6.1 One-step Sarsa Example

In this section, we illustrate Algorithm 9 by using the same example as in Section 5.3.1. The difference between these examples is that we now penalise the agent for not reaching the terminal state by giving it a reward of -1 instead of 0. Thus, the agent receives a reward of $+1$ when it reaches the terminal state and a reward of -1 otherwise.

We initialise the action-value function $Q(s, a)$ as zero for all state-action pairs (s, a). We start the first episode by initialising the state as $s = s_4$. The action we choose from the ϵ-greedy policy (where $\epsilon = 0.1$) is $a = a_1$. We observe the reward $r = -1$ and the next state $s' = s_3$. We choose the next action $a' = a_1$ from our ϵ-greedy policy and update the value estimate for $Q(s_4, a_1)$.

The next state-action pair to be evaluated is $(s, a) = (s_3, a_1)$. We take this action and observe the reward $r = -1$ and the next state $s' = s_2$. We again choose the next action as $a' = a_1$ from our ϵ-greedy policy. We update the value estimate for $Q(s_3, a_1)$ and set the next state-action pair to be evaluated as $(s, a) = (s_2, a_1)$.

We take this action and observe a reward of $r = -1$ and the next state as $s = s_1$. We choose the next action as $a' = a_1$ from our ϵ-greedy policy and update the value estimate for $Q(s_2, a_1)$.

We set the next state-action pair to be evaluated as $(s, a) = (s_1, a_1)$. We take this action and observe a reward of $r = +1$ and the next state as $s' = s_0$. Since $s' = s_0$ is the terminal state, we do not choose a next action a', but update the value estimate for $Q(s_1, a_1)$ (following the convention that $Q(s', a') = 0$ for all a' if s' is terminal) and then terminate the episode.

We follow the same train of thought for the rest of the simulation. Table 5-III shows the value estimates for $\alpha = 0.1$ after 60 episodes. The next method we will show converges much faster.

	Q		max	best
state	a_1	a_2	value	action
s_1	1	0	1	a_1
s_2	0	-1	0	a_1
s_3	-1	-1	-1	a_1 or a_2
s_4	-1	0	0	a_2
s_5	0	1	1	a_2

Table 5-III: Sarsa(0): Action-values after 60 episodes for $\alpha = 0.1$

From Table 5-III we can clearly see what the best actions would be in each situation. If you were in states s_1 or s_2, you would reach the terminal state fastest by walking left, *i.e.* taking action a_1. If you were in states s_4 or s_5, you would reach the terminal state fastest by walking right, *i.e.* taking action a_2. If you were in state s_3 (the middle of the grid), it would not matter which way you walked. The values correspond to the negative of the number of steps until termination, starting from that state. This is shown in Figure (5.2).

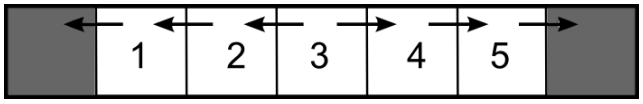


Figure 5.2: The optimal path

5.7 Q-learning: Off-policy TD control

Q-learning is an off-policy TD control algorithm that learns the expected values $Q(s,a)$ of taking action a in state s, then continues by always choosing actions optimally (Watkins and Dayan, 1992). In its simplest form, *one-step Q-learning*, is defined by

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)].$$

The learned action-value function Q, directly approximates Q^*, the optimal action-value function, independent of the policy being followed (Sutton and Barto, 1998a). This dramatically simplifies the analysis of the algorithm. The policy still has an effect in that it determines which state-action pairs are visited and updated. All that is required for correct convergence is that all pairs continue to be updated. Q_t has been shown to converge with probability one to Q^*. Q-learning is shown in procedural form in Algorithm 10.

Algorithm 10 Q-learning: Off-policy TD control algorithm

```
Initialize Q(s,a) arbitrarily
repeat
  (for each episode)
  Initialize s
  repeat
    (for each step of episode)
    Choose a from s using policy derived from Q (e.g. ε-greedy)
    Take action a, observe r, s'
    Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
    s ← s'
  until end of episode
until s is terminal
```

5.7.1 Q-learning Example

In this section, we illustrate Algorithm 10 by using the same example as in the previous section. The agent receives a reward of +1 when it reaches the terminal state and a reward of -1 otherwise.

We initialise the action-value function $Q(s, a)$ as zero for all state-action pairs (s, a). We start the first episode by initialising the state as $s = s_2$. The action we choose from the ϵ-greedy policy (where $\epsilon = 0.1$) is $a = a_2$. We observe the reward $r = -1$ and the next state $s' = s_3$. We choose the next action $a' = a_1$ from our ϵ-greedy policy and update the value estimate for $Q(s_2, a_2)$.

The next state-action pair to be evaluated is $(s, a) = (s_3, a_1)$. We take this action and observe the reward $r = -1$ and the next state $s' = s_2$. We again choose the next action as $a' = a_1$ from our ϵ-greedy policy. We update the value estimate for $Q(s_3, a_1)$ and set the next state-action pair to be evaluated as $(s, a) = (s_2, a_1)$.

We take this action and observe a reward of $r = -1$ and the next state as $s = s_1$. We choose the next action as $a' = a_1$ from our ϵ-greedy policy. We update the value estimate for $Q(s_2, a_1)$ and set the next state-action pair to be evaluated as $(s, a) = (s_1, a_1)$.

We take this action and observe a reward of $r = +1$ and the next state as $s' = s_0$. Since $s' = s_0$ is the terminal state, we do not choose a next action a', but update the value estimate for $Q(s_1, a_1)$ (following the convention that $Q(s', a') = 0$ for all a' if s' is terminal) and then terminate the episode.

Table 5-IV shows the value estimates for $\alpha = 0.1$ after 20 episodes.

	Q		max	best
state	a_1	a_2	value	action
s_1	1	0	1	a_1
s_2	0	-1	0	a_1
s_3	-1	-1	-1	a_1 or a_2
s_4	-1	0	0	a_2
s_5	0	1	1	a_2

Table 5-IV: $Q(0)$: Action-values after 20 episodes for $\alpha = 0.1$

From Table 5-IV we see what the best actions are for each state. If you were in states s_1 or s_2, you would reach the terminal state fastest by walking left, *i.e.* taking action a_1. If you were in states s_4 or s_5, you would reach the terminal state fastest by walking right, *i.e.* taking action a_2. If you were in state s_3 (the middle of the grid), it would not matter which way you walked. The $Q(0)$ algorithm obtains the same result as the one-step Sarsa algorithm, but much faster.

5.8 Concluding remarks

In this chapter we introduced TD learning. We reviewed on-policy and off-policy TD methods for solving the RL problem. The overall problem was divided into a prediction problem (policy evaluation) and a control problem (finding an optimal policy). TD methods were shown to be alternatives to MC methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of GPI.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (*e.g.* to be ϵ-greedy) with respect to the current value function. When the first process is based on experience, we have concerns regarding maintaining sufficient exploration. We grouped TD methods according to whether they deal with this complication or not by using an on-policy or off-policy approach. The on-policy method investigated was Sarsa, while the off-policy method was Q-learning.

TD methods are the most widely used RL methods. The reason for this is because of the great simplicity. It can be applied on-line, with a minimal amount of computation to experience generated from interaction with an environment and it can be expressed nearly completely by single equations. The TD methods discussed in this chapter can be called one-step, tabular and model free.

Chapter 6

Eligibility Traces

6.1 Introduction

So far we have discussed three classes of methods for solving the Reinforcement Learning (RL) problem: Dynamic Programming (DP), Monte Carlo (MC) and Temporal-Difference (TD) learning. Although each method is different, there is no need to pick just one as it is often desirable to apply different kinds of methods at the same time. We now develop specific algorithms that embody the core ideas of one or more of the elementary solution methods presented in Chapters 3-5 and look at unifying these methods.

In this chapter we show how eligibility traces with TD errors provide an efficient way of shifting and choosing between MC and TD methods. We look at the theory of eligibility traces and then at n-step TD prediction. We also consider the forward and backward view of TD(λ) and establish the ways in which they are equivalent. We first consider the prediction problem and then the control problem and only after exploring the two views of eligibility traces within the prediction setting do we extend the ideas to action values and control methods, Sarsa(λ) and $Q(\lambda)$, and then look at replacing traces.

6.2 Theory of eligibility traces

In the TD(λ) algorithm, the λ refers to the use of an eligibility trace. Almost any TD method can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

There are two ways to view eligibility traces, namely *theoretical* and *mechanistic* (Sutton and Barto, 1998*a*). The theoretical view is that eligibility traces are a bridge from TD to MC methods. When TD methods are expanded with eligibility traces, they produce a family of

methods spanning a spectrum that has MC methods at one end and one-step TD methods at the other.

The mechanistic view is that an eligibility trace is a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action. The trace flags the memory parameters associated with the event, indicating that they are suitable, or *eligible*, for undergoing learning changes. When a TD error occurs, only the flagged states or actions are assigned credit or blame for the error.

The theoretical view can also be thought of as the *forward* view and the mechanistic view as the *backward* view. The forward view is especially useful for understanding what is computed by methods using eligibility traces while the backward view is more appropriate for developing intuition about the algorithms themselves.

6.3 n-Step TD prediction

Suppose we estimate V^π from sample episodes generated by π. Suppose further that we consider the problem from a theoretical point of view. At the one end of the spectrum, MC performs a backup for each state based on the entire sequence of observed rewards from that state until the end of the episode. At the other end of the spectrum, a simple TD backup is based on just the one next reward, using the value of the state one step later as a proxy for the remaining rewards. An intermediate method would perform a backup based on an intermediate number of rewards, *i.e.* more than one, but less than all of them. For example, a two-step backup would be based on the first two rewards and the estimated value of the state two steps later. An n-step backup would be based on the first n rewards and the estimated value of the state n steps later. Methods that use n-step backups are still TD methods: they still change an earlier estimate based on how it differs from a later estimate. The later estimate is now not one step later, but n steps later. Methods in which the temporal-difference extends over n steps are called *n-step TD methods* (Sutton and Barto, 1998*a*). From now on we will refer to the TD methods in Chapter 5 as *one-step TD methods*.

The reader is referred to Appendix C.2 for a derivation of the n-step target, as well as the n-step return, n-step backups and an explanation of why n-step TD methods form a family of valid methods, with one-step TD methods and MC methods as extreme members.

n-Step TD methods are rarely used because they are inconvenient to implement. Computing n-step returns requires waiting n steps to observe the resultant rewards and states. This can become problematic for large n, thereby rendering the significance of n-step TD as primarily theoretical.

6.4 Forward view of TD(λ)

We can apply backups towards any average of n-step returns, not just towards an n-step return. For example, a backup can be done towards a return that is half of a one-step return and half of a three-step return: $R_t^{\text{ave}} = \frac{1}{2}R_t^{(1)} + \frac{1}{2}R_t^{(3)}$. Any set of returns can be averaged in this way, as long as the weights on the component returns are positive and sum to one. The overall return can be used to construct backups with guaranteed convergence properties, as it has an error reduction property similar to that of individual n-step returns. Averaging produces a new range of algorithms, *e.g.* one could average one-step and infinite-step backups to obtain another way of merging TD and MC methods.

The TD method has been generalised to TD(λ), where $0 \leq \lambda \leq 1$ is a weighting on the applicability of temporal differences of previous predictions (Engelbrecht, 2007). The TD(λ) algorithm is one way of averaging n-step backups. It contains all the n-step backups, each weighted proportional to λ^{n-1}. The normalisation factor of $1-\lambda$ ensures that weights sum to one. The resulting backup is towards a return, called the λ-*return* (Sutton and Barto, 1998*a*). This is defined by

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)}. \tag{6.1}$$

The one-step return is given the largest weight, $1-\lambda$. The two-step return is given the second largest weight, $(1-\lambda)\lambda$; the three-step return the third largest, $(1-\lambda)\lambda^2$, and so on. The weight is reduced by λ with each additional step. After reaching a terminal state, all next n-step returns are equal to the return R_t. These terms can be separated from the main sum:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}R_t^{(n)} + \lambda^{T-t-1}R_t. \tag{6.2}$$

If $\lambda = 1$, then the summation term goes to zero and the remaining term reduces to the return, R_t. Thus for $\lambda = 1$, backing up according to the λ-return is the same as the MC algorithm, constant-α MC. If $\lambda = 0$, then the λ-return reduces to $R_t^{(1)} = \sum_{n=1}^{T-t-1} R_t^{(n)}$, the one-step return. Thus for $\lambda = 0$, backing up according to the λ-return is the same as the one-step TD method, TD(0).

The λ-return algorithm is defined as the algorithm that performs backups using the λ-return. On each step t, it computes an increment $\Delta V_t(s_t)$ to the value of the state occurring at that step:

$$\Delta V_t(s_t) = \alpha[R_t^{\lambda} - V_t(s_t)].$$

The increments for the other states are $\Delta V_t(s) = 0 \; \forall \; s \neq s_t$. Updating can be either on-line or off-line.

For each state visited, we look forward in time to all future rewards and decide what is the best way to combine them. After looking forward from and updating one state, we move on

to the next state and never have to work with the previous state again.

The λ-return algorithm is the basis for the forward view of eligibility traces as used in the TD(λ) algorithm. In the off-line case, the λ-return algorithm is the TD(λ) algorithm. The λ-return and TD(λ) methods use the λ parameter to shift from one-step TD to MC.

6.5 Backward view of TD(λ)

In the previous section we presented the forward, or theoretical, view of tabular TD(λ) as a way of mixing backups that parametrically shift from a TD method to an MC method. Here, we define TD(λ) mechanistically. The mechanistic, or backward, view of TD(λ) is useful because it is relatively easy to understand and relatively easy to implement. The forward view, on the other hand, is not directly implementable because it uses knowledge at each step of what will happen many steps later.

We now introduce the concept of an eligibility trace, which is a memory variable associated with each state in the backward view of TD(λ). An eligibility trace, $e_t(s) \in \Re^+$, is defined as the eligibility trace for state s at time t. On each step, eligibility traces for all states decay by $\gamma\lambda$. The eligibility trace for the state visited on the step is incremented by one:

$$e_t(s) = \begin{cases} \gamma\lambda\, e_{t-1}(s), & \text{if } s \neq s_t \\ \gamma\lambda\, e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{6.3}$$

for all $s \in S$, where γ is the discount rate. From here on we refer to λ as a *trace-decay parameter*. This kind of eligibility trace is called an *accumulating trace*, because it accumulates each time a state is visited, then fades away gradually when the state is not visited.

At any time, eligibility traces record which states have recently been visited, where "recently" is defined in terms of $\gamma\lambda$. Traces indicate the degree to which each state is eligible for undergoing learning changes, should a reinforcing event occur. Reinforcing events are thought of as moment-by-moment one-step TD errors.

Algorithm 11 shows the complete outline of the on-line tabular TD(λ) algorithm.

The backward view of TD(λ) is oriented backwards in time. At each moment we look back at the current TD error and assign it backwards to each prior state according to the state's eligibility trace at that time. Where the TD error and the traces come together, we get the following update:

$$\Delta V_t(s) = \alpha\delta_t e_t(s),\ \forall s \in S, \tag{6.4}$$

where

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t). \tag{6.5}$$

Algorithm 11 On-line tabular TD(λ)

Initialise V(s) arbitrarily for all $s \in \mathcal{S}$
repeat
  (for each episode)
  Initialise $e(s) = 0$ for all $s \in \mathcal{S}$
  Initialise s
  repeat
    (for each step of episode)
    $a \leftarrow$ action given by π for s
    Take action a, observe reward r and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for all s **do**
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    end for
    $s \leftarrow s'$
  until end of episode
until s is terminal

Consider what happens at various values of λ. If $\lambda = 0$, then by equation (6.3) all traces are zero at t except for the trace corresponding to s_t. Thus the TD(λ) update equation (6.4) reduces to the simple TD rule, TD(0). TD(0) is the case in which only the one state preceding the current one is changed by the TD error. For larger values of λ (but still $\lambda < 1$), more of the preceding states are changed, but each more temporally distant state is changed less because its eligibility trace is smaller. Earlier states are given less credit for the TD error.

If $\lambda = 1$, the credit given to earlier states falls only by γ per step. If $\lambda = \gamma = 1$, then the eligibility traces do not decay at all with time. In this case the method behaves like a MC method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is known as TD(1).

TD(1) is a way of implementing MC that is more general than those presented earlier, thereby increasing the range of applicability. While earlier MC methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. TD(1) can be performed incrementally and on-line. On-line TD(1) learns in an n-step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusual happens during an episode, control methods based on TD(1) can learn immediately and alter their behaviour on that same episode.

The reader is referred to Appendix C.1 for an explanation of the equivalence of forward and backward views.

6.5.1 TD(λ) Example

We illustrate Algorithm 11 with the same example as in Section 5.3.1. To refresh the reader's memory we show a graphical representation in Figure (6.1).

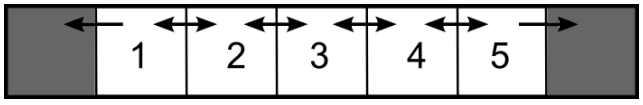


Figure 6.1: The simple walk revisited

We initialise $V(s)$ as zero for all states $s \in \mathcal{S}$ and randomly visit all the states a number of times. Note that in this example, just as in the TD(0) example, the agent is given a reward of $+1$ when it reaches the terminal state and 0 otherwise.

Episode 1

We initialise state $s = s_1$ and the eligibility traces $e(s) = 0$ for all states. Action $a = a_1$ is given by our policy π for s. After taking this action, we observe that the reward is $r = +1$ and the next state $s' = s_0$. This is illustrated in Figure (6.2).

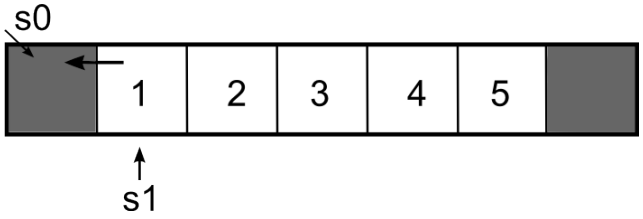


Figure 6.2: First episode

We calculate δ and the eligibility trace for this particular state and find that $\delta \leftarrow 1$ and $e(s_1) \leftarrow 1$, where δ is the TD error for the state-value prediction. This TD error triggers proportional updates to all recently visited states. The eligibility trace $e(s)$ records which states have recently (in terms of $\gamma\lambda$) been visited. The trace indicates which state is *eligible* for undergoing learning changes, should a certain event occur. We then update the value function and eligibility traces for every state. It is only necessary to compute these updates for the states that were changed so far in the episode. A state that was not visited has an eligibility trace of zero, thus calculating $\alpha\delta e(s) \leftarrow 0$, and therefore $V(s) \leftarrow V(s)$. For the same reason the new eligibility trace will also be zero, $e(s) \leftarrow \gamma\lambda e(s)$, thereby rendering it pointless to recalculate it.

The only state that was visited so far is s_1 and its value function and eligibility trace is found to be

$$\begin{aligned} V(s_1) &\leftarrow \alpha \\ e(s_1) &\leftarrow \lambda, \end{aligned}$$

where α is again a constant stepsize parameter.

State s_0 is then set as the state to be visited next, but s_0 is the terminal state, thus the first episode ends.

Episode 2

For the second episode we reset the eligibility traces to zero and initialised the state as $s = s_2$. Action $a = a_1$ was given by policy π for s. We observed the next state as $s' = s_1$ and the reward as $r = +0$. The observation of the next state can be seen in Figure (6.3).

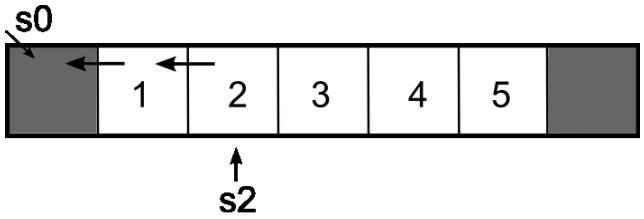


Figure 6.3: Second episode

Calculations were made in the same fashion as for the first episode. The order in which the states and actions were taken is: $s_2 \rightarrow a_1 \rightarrow s_1 \rightarrow a_1 \rightarrow s_0$. The following updates were made at the end of the second episode:

$$\begin{aligned} V(s_1) &\leftarrow \alpha + \alpha(1-\alpha) \\ e(s_1) &\leftarrow \lambda \\ V(s_2) &\leftarrow \alpha^2 + \alpha\lambda(1-\alpha) \\ e(s_2) &\leftarrow \lambda^2. \end{aligned}$$

Episode 3

For the third episode the eligibility traces were again reset to zero and we initialised s_3 as the new state. This is shown in Figure (6.4).

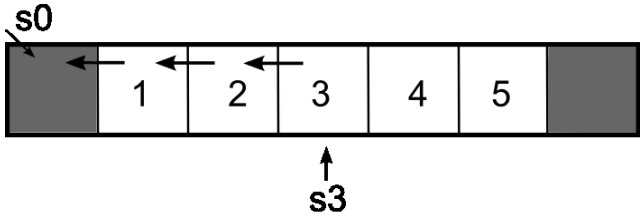


Figure 6.4: Third episode

The order in which the states and actions were taken is: $s_3 \to a_1 \to s_2 \to a_1 \to s_1 \to a_1 \to s_0$. The following updates were made at the end of the third episode:

$$\begin{aligned}
V(s_1) &\leftarrow \alpha + \alpha(1-\alpha) + \alpha(\alpha-1)^2 \\
e(s_1) &\leftarrow \lambda \\
V(s_2) &\leftarrow \alpha^2 + \alpha\lambda(1-\alpha) + \alpha^2(1-\alpha)(2-\lambda) + \alpha\lambda(\alpha-1)^2 \\
e(s_2) &\leftarrow \lambda^2 \\
V(s_3) &\leftarrow \alpha^2(\alpha + \lambda(1-\alpha)) + \alpha^2\lambda(1-\alpha)(2-\lambda) + \alpha\lambda^2(\alpha-1)^2 \\
e(s_3) &\leftarrow \lambda^3
\end{aligned}$$

Episode 4

In the fourth episode we observed action a_2, instead of a_1 as we have thus far. The observation of the next state can be seen in Figure (6.5).

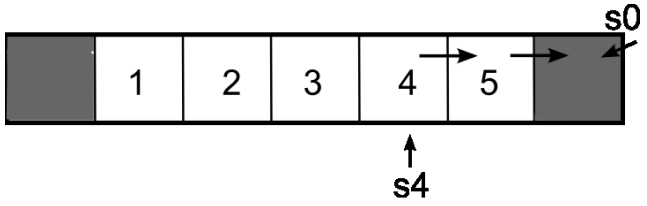


Figure 6.5: Fourth episode

The order in which the states and actions were taken is: $s_4 \to a_2 \to s_5 \to a_2 \to s_0$. The first three states were not used in this episode, and so the updates made are only concerning states s_4 and s_5:

$$\begin{aligned}
V(s_4) &\leftarrow \alpha\lambda \\
e(s_4) &\leftarrow \lambda^2 \\
V(s_5) &\leftarrow \alpha \\
e(s_5) &\leftarrow \lambda.
\end{aligned}$$

Episode 5

In the fifth and final episode that we discuss, we again observe action a_2 and it is only state s_5 that is changed in the episode. This is illustrated in Figure (6.6).

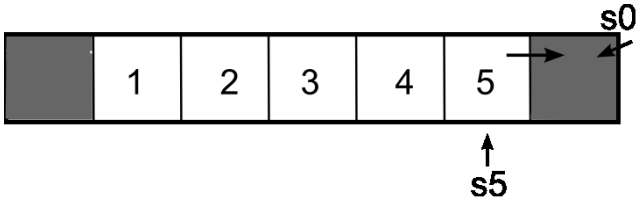


Figure 6.6: Fifth episode

The order in which the states and actions were taken is: $s_5 \rightarrow a_2 \rightarrow s_0$. The following updates were made:

$$\begin{aligned} V(s_5) &\leftarrow \alpha + \alpha(1-\alpha) \\ e(s_5) &\leftarrow \lambda. \end{aligned}$$

Results

We have now visited each state exactly once and updated them as we went along. The complete value function $V(s)$ is now:

$$\begin{aligned} V(s_1) &= \alpha(3 - 3\alpha + \alpha^2) \\ V(s_2) &= \alpha(\alpha(3-4\lambda) + 2\alpha^2(\lambda - 1) + 2\lambda) \\ V(s_3) &= \alpha^2(\alpha + \lambda(1-\alpha)) + \alpha^2\lambda(1-\alpha)(2-\lambda) + \alpha\lambda^2(\alpha-1)^2 \\ V(s_4) &= \alpha\lambda \\ V(s_5) &= \alpha(2-\alpha) \end{aligned}$$

We took these equations and substituted for α and λ values. These estimated values can be seen in Table 6-I for $\alpha = 0.1$ at the end of the fifth episode.

If $\lambda = 0$, then all traces in an episode are zero at t except for the trace corresponding to s_t. The TD(λ) update,

$$\Delta V_t(s) = \alpha\delta_t e_t(s),$$

reduces to the simple TD(0) rule,

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)],$$

λ	$V(s_1)$	$V(s_2)$	$V(s_3)$	$V(s_4)$	$V(s_5)$
0.0	0.271	0.028	0.001	0.000	0.190
0.1	0.271	0.044	0.004	0.010	0.190
0.2	0.271	0.060	0.009	0.020	0.190
0.3	0.271	0.077	0.016	0.030	0.190
0.4	0.271	0.093	0.023	0.040	0.190
0.5	0.271	0.109	0.033	0.050	0.190
0.6	0.271	0.125	0.043	0.060	0.190
0.7	0.271	0.141	0.055	0.070	0.190
0.8	0.271	0.158	0.069	0.080	0.190
0.9	0.271	0.174	0.084	0.090	0.190
1.0	0.271	0.190	0.100	0.100	0.190

Table 6-I: TD(λ): Estimated value function after one run

where TD(0) is the case in which only the state preceding the current one is changed by the TD error.

For $0 < \lambda \leq 1$, more of the earlier states are changed, but each more temporally distant state is changed less because its eligibility trace is smaller.

If $\lambda = 1$, credit given to earlier states falls only by γ per step. This is the right thing to do to achieve MC behaviour and this specific algorithm is also known as TD(1). The TD error, δ_t, includes an undiscounted term of r_{t+1}. Passing this back k steps it needs to be discounted by γ^k. This is exactly what the falling eligibility trace achieves. If both $\gamma = 1$ and $\lambda = 1$, then $e(s) \leftarrow e(s)$ and thus the traces do not decay at all with time. In this case the methods behave like a MC method for an undiscounted, episodic task, which is exactly what we have here.

6.6 Sarsa(λ)

When we want to find optimal policies, we refer to it as the *control problem*. So the next question we ask is, how can eligibility traces be used for control? One popular approach is to learn action values $Q_t(s, a)$, rather than state values $V_t(s)$ just as in Section 5.6. We now combine eligibility traces with Sarsa to produce an on-policy TD control method.

The eligibility trace version of Sarsa is called *Sarsa(λ)*. From this point forward we will refer to the original version presented in Section 5.6 as *one-step Sarsa*.

The idea in Sarsa(λ) is to apply the TD(λ) prediction method to state-action pairs rather than to states. We need a trace for each state-action pair. Let $e_t(s, a)$ denote the trace for the state-action pair (s, a). Other than that the method is just like TD(λ), substituting state-action variables for state variables: $Q(s, a)$ for $V(s)$ and $e_t(s, a)$ for $e_t(s)$, providing us with:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a), \ \forall \ s, a$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t),$$

and

$$e_t(s,a) = \begin{cases} \gamma\lambda\, e_{t-1}(s,a) + 1, & \text{if } s = s_t, a = a_t \\ \gamma\lambda\, e_{t-1}(s,a), & \text{otherwise.} \end{cases}$$

The backup for Sarsa(λ) is similar to TD(λ). The first backup looks ahead one full step to the next state-action pair, the second looks ahead two steps, and so on. The final backup is based on the complete return. The weighting of each backup is just as in TD(λ) and the λ-return algorithm.

One-step Sarsa and Sarsa(λ) are on-policy algorithms; they approximate $Q^\pi(s,a)$, the action values for the current policy. Policy improvement can be done in many different ways. One of the most simple approaches is to use the ϵ-greedy policy with respect to the current action-value estimates. Algorithm 12 gives the complete outline for the tabular Sarsa(λ) algorithm.

Algorithm 12 Tabular Sarsa(λ)

Initialise $Q(s,a)$ arbitrarily for all s, a
repeat
  (for each episode)
  Initialise $e(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
  Initialise s, a
  repeat
    (for each step of episode)
    Take action a, observe reward r and next state s'
    Choose a' from s' using policy derived from Q
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all s, a **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    end for
    $s \leftarrow s'; a \leftarrow a'$
  until end of episode
until s is terminal

6.6.1 Sarsa(λ) Example

We illustrate Algorithm 12 with the same example as in Section 5.6.1. The idea in Sarsa(λ) is to apply the TD(λ) prediction method to state-action pairs as opposed to just states.

This means we need a trace for each state-action pair and not just for each state. Sarsa(λ) is just like TD(λ), only substituting state-action variables for state variables.

To illustrate the algorithm, we initialise $Q(s, a)$ as zero for all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$ and randomly traverse through all the states. The agent receives a reward of +1 when it reaches the terminal state and a reward of -1 otherwise.

We start the first episode by initialising the eligibility trace $e(s, a) = 0$ for all states s and actions a. We choose the starting state as $s = s_3$ and the action as $a = a_1$. We take this action and observe the reward $r = -1$ and the next state $s' = s_2$. We choose the next action $a' = a_1$ from our ϵ-greedy policy (where $\epsilon = 0.1$). We calculate δ and update the trace $e(s_3, a_1)$. For all states and actions we update the value estimate for $Q(s, a)$ and the trace $e(s, a)$.

The next state-action pair to be evaluated is $(s, a) = (s_2, a_1)$. We take this action and observe the reward $r = -1$ and the next state $s' = s_1$. We again choose the next action as $a' = a_1$ from our ϵ-greedy policy. We calculate δ and $e(s_2, a_1)$ and use this to update the value estimate $Q(s, a)$ for all states and actions. We set the next state-action pair to be evaluated as $(s, a) = (s_1, a_1)$.

We take this action and observe a reward of $r = +1$ and the next state as $s' = s_0$. Since $s' = s_0$ is the terminal state, we do not choose a next action a'. Again we use the convention that $Q(s', a') = 0$ for all a' if s' is terminal. We calculate δ and $e(s_1, a_1)$, update the value estimate $Q(s, a)$ for all states and actions and then terminate the episode.

We follow the same train of thought for the rest of the simulation. Table 6-II shows the value estimates for $\alpha = 0.1$ after 60 episodes.

	Q		max	best
state	a_1	a_2	value	action
s_1	1	0	1	a_1
s_2	0	-1	0	a_1
s_3	-1	-1	-1	a_1 or a_2
s_4	-1	0	0	a_2
s_5	0	1	1	a_2

Table 6-II: Sarsa(λ): Action-values after 60 episodes for $\alpha = 0.1$

From Table 6-II we can clearly see what the best actions would be in each situation. If you were in states s_1 or s_2, you would reach the terminal state fastest by walking left, *i.e.* taking action a_1. If you were in states s_4 or s_5, you would reach the terminal state fastest by walking right, *i.e.* taking action a_2. If you were in state s_3 (the middle of the grid), it would not matter which way you walked.

6.7 Watkins's Q(λ)

In this section we propose a method that combines eligibility traces and Q-learning called *Watkins's* $Q(\lambda)$. In Chapter 5.7 we saw that Q-learning is an off-policy method where the policy learned about is not necessarily the same as the one being followed. Q-learning learns a greedy policy while it follows one that occasionally explores other actions that are suboptimal according to Q_t (Sutton and Barto, 1998*a*).

Suppose that we want to update the state-action pair (s_t, a_t) at time t. On the next two time steps ($t+1$ and $t+2$) we take a greedy action, but on the third time step ($t+3$) we take an exploratory, non-greedy action. In estimating the value function at (s_t, a_t), we can only use the experience obtained by following the greedy policy. Thus, we can only use the one-step and two-step returns, but not the three-step return. In this case, the n-step returns for all $n \geq 3$ no longer have any relevant relationship to the greedy policy.

Watkins's $Q(\lambda)$ does not look ahead all the way to the end of the episode in its backup. It only looks ahead as far as the next exploratory action or at the episode's end if there are no exploratory actions before that. Watkins's $Q(\lambda)$ looks one action past the first exploration, using its knowledge of the action values.

The mechanistic, or backward, view of Watkins's $Q(\lambda)$ is very simple. Eligibility traces are used just as in Sarsa(λ), except that they are set to zero whenever an exploratory (non-greedy) action is taken. The trace update is best thought of as occurring in two steps. First, the traces for all state-action pairs are either decayed by $\gamma\lambda$ or, if an exploratory action was taken, set to zero. Second, the trace corresponding to the current state and action is incremented by one. The overall result is:

$$e_t(s,a) \quad = \quad \mathcal{I}_{ss_t} \cdot \mathcal{I}_{aa_t} + \begin{cases} \gamma\lambda\, e_{t-1}(s,a), & \text{if } Q_{t-1}(s_t,a_t) = \max\limits_a Q_{t-1}(s_t,a) \\ \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{I}_{xy}$ is the identity-indicator function, equal to one if $x = y$ and zero if $x \neq y$. The rest of the algorithm is defined by

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a),$$

where

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t).$$

Algorithm 13 gives the complete outline of the tabular version of Watkins's $Q(\lambda)$ algorithm.

Cutting off traces every time an exploratory action is taken, loses much of the advantage of using eligibility traces. If exploratory actions are frequent, then only rarely will backups of

Algorithm 13 Tabular version of Watkins's $Q(\lambda)$ algorithm

Initialise $Q(s,a)$ arbitrarily for all s, a
repeat
  (for each episode)
  Initialise $e(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
  Initialise s, a
  repeat
    (for each step of episode)
    Take action a, observe reward r and next state s'
    Choose a' from s' using policy derived from Q
    $a^* \leftarrow \arg\max_b Q(s', b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
    $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all s, a **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      if $a' = a^*$ **then**
        $e(s,a) \leftarrow \gamma\lambda e(s,a)$
      else
        $e(s,a) \leftarrow 0$
      end if
    end for
    $s \leftarrow s'; a \leftarrow a'$
  until end of episode
until s is terminal

more than one or two steps be done.

6.7.1 $Q(\lambda)$ Example

We illustrate Algorithm 13 with the same example as in the previous section. We initialize $Q(s,a)$ as zero for all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$ and randomly visit all the states a number of times. The agent receives a reward of +1 when it reaches the terminal state and a reward of -1 otherwise.

We start the first episode by initializing the eligibility trace $e(s,a) = 0$ for all states s and actions a. We choose the starting state as $s = s_2$ and the action as $a = a_1$. We take this action and observe the reward $r = -1$ and the next state $s' = s_1$. We choose the next action $a' = a_1$ from our ϵ-greedy policy (where $\epsilon = 0.1$). We calculate a^* and δ and update the trace $e(s_2, a_1)$. For all states and actions we update the value estimate for $Q(s,a)$ and the trace $e(s,a)$.

The next state-action pair to be evaluated is $(s,a) = (s_1, a_1)$. We take this action and observe a reward of $r = +1$ and the next state as $s' = s_0$. Since $s' = s_0$ is the terminal state, we do not choose a next action a', nor calculate a^*. Again we use the convention that $Q(s', a') = 0$ for all a' if s' is terminal. We calculate δ (here also assuming that $Q(s', a^*) = 0$ if s' is terminal) and $e(s_1, a_1)$, update the value estimate $Q(s,a)$ for all states and actions

and then terminate the episode.

We follow the same train of thought for the rest of the simulation. Table 6-III shows the value estimates for $\alpha = 0.1$ after 50 episodes.

	Q		max	best
state	a_1	a_2	value	action
s_1	1	0	1	a_1
s_2	0	-1	0	a_1
s_3	-1	-1	-1	a_1 or a_2
s_4	-1	0	0	a_2
s_5	0	1	1	a_2

Table 6-III: $Q(\lambda)$: Action-values after 50 episodes for $\alpha = 0.1$

From Table 6-III we can clearly see what the best actions would be in each situation. If you were in states s_1 or s_2, you would reach the terminal state fastest by walking left, *i.e.* taking action a_1. If you were in states s_4 or s_5, you would reach the terminal state fastest by walking right, *i.e.* taking action a_2. If you were in state s_3 (the middle of the grid), it would not matter which way you walked. Again we see that the Q-learning algorithm is faster than the Sarsa algorithm, even though it is only slightly faster in the case where eligibility traces are used.

6.8 Peng's $Q(\lambda)$

An alternate version of $Q(\lambda)$, called *Peng's* $Q(\lambda)$, can be thought of as a hybrid of Sarsa(λ) and Watkins's $Q(\lambda)$ (Sutton and Barto, 1998*a*).

Peng's $Q(\lambda)$ uses a mixture of backups. Unlike Q-learning, there is no distinction between exploratory and greedy actions. Each component backup is over many steps of actual experience, and all but the last are capped by a final maximisation over actions causing the component backups to be neither on-policy nor off-policy. Earlier transitions of each are on-policy, whereas the last (fictitious) transition uses the greedy policy. As a consequence, for a fixed non-greedy policy, Q_t converges to some hybrid of Q^π and Q^*. If the policy is gradually made more greedy, then the method may still converge to Q^*. It performs better than Watkins's $Q(\lambda)$ and almost as well as Sarsa(λ), but according to (Leng *et al.*, 2009) it "is difficult to implement, and is of no real implementation."

6.9 Replacing traces

In certain cases, such as off-line applications in which data can be generated cheaply, performance can be greatly improved by using replacing traces. Suppose a state is visited and then revisited before the trace due to the first visit has fully decayed to zero. With accumulating

traces, the revisit causes a further increment in the trace, driving the trace greater than one. With replacing traces, the trace is set to one. The replacing trace for a discrete state s is defined by:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & \text{if } s \neq s_t \\ 1 & \text{otherwise.} \end{cases}$$

Prediction or control algorithms using replacement traces are often called *replace-trace methods*. Although replacing traces are only slightly different from accumulating traces, they can produce a significant improvement in the learning rate.

There are several ways to generalise replacing eligibility traces for use in control methods. When a state is revisited and a new action is selected, the trace for that action should be reset to one. What about the traces for the other actions for that state? The recommended approach is to set the traces of all the other actions from the revisited state to zero. In this case, the state-action traces are updated in the following way:

$$e_t(s,a) = \begin{cases} 1, & \text{if } a = a_t, s = s_t \\ 0, & \text{if } a \neq a_t, s = s_t \\ \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t \end{cases}$$

6.10 Implementation issues

It might seem that methods using eligibility traces are much more complex than one-step methods, but for typical values of λ and γ the eligibility traces of almost all states are almost always nearly zero. Only those that have recently been visited will have traces significantly greater than zero. Only these few states really need to be updated because the updates at the others will have no effect.

In practice, implementations on computers keep track of and update only the few states with non-zero traces. Using this trick, the computational expense of using traces is usually a few times that of a one-step method. The exact multiple depends on λ and γ and on the expense of the other computations.

6.11 Concluding remarks

In this chapter we saw how eligibility traces can be used together with TD methods and MC methods to obtain more efficient learning.

Eligibility traces with TD errors provide an efficient way of shifting between MC and TD methods. Traces can be used without TD errors to achieve a similar effect, but only awkwardly. A method such as TD(λ) enables this to be done from partial experiences and with little memory.

Although we do not assume non-Markov tasks, it is interesting to note that MC methods may have advantages in non-Markov tasks because they do not bootstrap. Because eligibility traces make TD methods more like MC methods, they can also have advantages in these cases. If you want to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated. Eligibility traces are the first line of defence against both long-delayed rewards and non-Markov tasks.

On tasks with many steps per episode, it appears significantly better to use eligibility traces than not to. On the other hand, if the traces are so long as to produce a pure MC method, then performance degrades sharply. An intermediate mixture appears to be the best choice. Eligibility traces should be used to bring us toward MC methods, but not all the way there.

Methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. If often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed. However, in off-line applications in which data can be generated cheaply, it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as quickly as possible. In these cases the speedup due to traces is typically not worth their computational cost and one-step methods are then favoured.

Chapter 7

Threat Evaluation and Weapon Assignment

7.1 Introduction

The increase in sophistication of threats within the military environment necessitates an increase in Situation Awareness (SA) and combat readiness in defensive operations (Potgieter, 2007). Ever advancing technology enables the integration of military forces and systems, operating in different environments, to combine their strengths and cooperate toward a common goal.

An air defence (AD) operator is required to evaluate the tactical situation in real-time and protect defended assets (DAs) against aerial threats by assigning available weapon systems to engage enemy aircraft (Project TEWA). Since this aerial environment requires rapid operational planning and decision making in stress situations, the associated responsibilities are typically divided between a number of operators and computerised systems that aid these operators during the decision making processes. Threat Evaluation and Weapon Assignment (TEWA) is one such a Decision Support System (DSS), assigning threat values to aircraft (with respect to DAs) in real-time and using these values to suggest possible assignments between anti-aircraft weapons and observed enemy aircraft.

The TEWA process may be regarded as a dynamic human decision making process aimed at the successful exploitation of tactical resources (*i.e.* sensors, weapons) during the conduct of Command and Control (C2) activities (Paradis *et al.*, 2005). From this perspective, that reflects today's reality, the role of humans is central to the TEWA functions as they are responsible for any outcome resulting from their decisions. Hence, the type of support or aid to be provided is meant to assist, not replace, operators in their decision making activities. As a result, the automation to be provided requires careful design. In order to provide this

type of automation, an approach is required that models the decision making problem and captures the cognitive demands and information requirements. The local TEWA environment is defined for AD, with initial focus on Ground Base Air Defence Systems (GBADS) (Leenen and Hakl, 2009) and consists of two independent specialised subsystems in which the Threat Evaluation (TE) and Weapon Assignment (WA) computations are executed (Potgieter, 2007).

In the remainder of this chapter we explain the relevant components of TEWA, and briefly describe their workings. We note the difference between the two subsystems, TE and WA, and explain why our focus is on WA, rather than on TE.

7.2 TEWA

7.2.1 Air defence

Advances in technology and research have increased the accuracy and effectiveness of AD systems over the years (Potgieter, 2007). At the turn of the 20^{th} century AD systems were employed by almost every military organisation and are primarily required to protect assets from aerial threats, including manned aerial vehicles, unmanned aerial vehicles, space vehicles, cruise missiles, ballistic missiles and other emerging threats. AD systems employed for this purpose are called *strategic AD systems* and are usually large, technologically advanced and mostly fixed installations. These systems are employed for the protection of operational surfaces and air forces from aerial attack.

7.2.2 Ground Based Air defence

In the Ground Based Air defence (GBAD) context, there are a number of prioritised DAs, ranging from fixed installations (such as factories, bridges and hangars) to moving objects (such as battleships, convoys and brigades) that are at risk of being targeted by enemy aircraft and need protection (Potgieter, 2007). A well-established network of sensors, communications and weapon systems may be deployed in the immediate tactical area to counter these threats.

All the elements may be operated remotely from a Command and Control (C2) centre connected to the network via various data and communication links (*e.g.* LANs (local area networks) or by Radio Frequency (RF)). A GBADS operator is responsible for optimally allocating the available weapon systems to engage enemy aircraft within a limited space of time and under severe stress. This allocation problem may become rather complex as the number of weapon systems and the number of enemy aircraft increase and is almost impossible for a human to solve optimally (or even approximately) in real-time without computational aid.

Current GBAD systems may be divided into three main categories, namely artillery systems, missile systems and laser weapons systems.

Artillery systems

Artillery systems may be divided into two classes namely towed and static artillery, and self-propelled artillery (Potgieter, 2007). Towed artillery are typically slower, being pulled behind a truck or tractor, and may be employed together with self-propelled artillery to accomplish a more diverse arsenal.

Self-propelled artillery are able to reposition and set up more quickly, but are heavier. Self-propelled anti-aircraft (AA) guns typically consist of a light artillery cannon mounted on an armoured military vehicle which may additionally be equipped with surveillance, radar, communication and other electronic systems. These AA guns are radar guided and are usually paired to increase their effectiveness. Larger AA guns may be deployed in a fixed location or occasionally towed for redeployment when necessary. These systems must be stationary to be operational.

Missile systems

Surface-to-Air Missiles (SAMs) may be divided into the three classes of man-portable, self-propelled and static and towed SAMs. Man-portable SAMs are typically small, mobile units used for Very SHOrt Ranged Air defence (VSHORAD) (Potgieter, 2007). Self-propelled SAMs are typically longer range, but are not capable of reaching and deploying in all the areas in which a man-portable SAM is able to deploy. Static and towed SAMs are typically large, relatively stationary, longer range weapon systems.

Man-portable SAMs are shoulder-launched missile systems designed to attack low-flying aerial threats. They typically make use of infra-red guidance systems and AA guided missiles and may be operated by a single person.

A self-propelled SAM system typically comprises one or more SAMs mounted on a military vehicle which may be equipped with surveillance, radar, communication and other electronic systems. Larger SAM systems may be deployed in a fixed location or occasionally towed for redeployment when required. However, these systems must be stationary when operational and are typically guided by radar.

Laser weapon systems

Future AD systems may include the use of directed energy weapon systems such as lasers. These AD systems are expected to be able to destroy approaching airborne threats in midair,

including late detection threats (such as artillery rockets), high-speed threats (such as short-range ballistic and cruise missiles) and other threats (such as Unmanned Aerial Vehicles (UAVs)).

7.2.3 The role of the operator

Operators may experience an overload of information without the incorporation of DSSs for critical real-time decision making. Data collected by a number of sensors are transmitted on a real-time basis to other nodes networked together, typically sharing a common, complex picture of the battle space. Operators responsible for these nodes are required to decide on a course of action in real-time and under pressure, based on the information available.

With the introduction of DSSs, an operator's task has shifted more towards the monitoring of the automated DSS instead of performing all the tasks manually. The quality of the DSS process, however, depends on the accuracy of the input information, the quality of prediction algorithms and the depth of the DSS software implementation. An operator must draw upon his/her available knowledge, experience and awareness factors that may not be considered by the DSS to decide whether or not to accept the DSS output, and therefore requires a thorough understanding of the characteristics, behaviour and limitations of the DSS. Reasons for intervention may include perceived deception, prior knowledge of expected threat behaviour, reports from Observation Posts (OPs), own system status, appreciation of risk and environmental influences.

7.2.4 The tactical environment

The *tactical environment* refers to the physical real-world conditions within which (enemy and friendly) units and systems operate. Environmental conditions such as different terrain features and weather conditions, typically affect the performance and deployment of sensor systems, the performance and deployment of weapon systems as well as the performance and behaviours of attacking enemy aircraft. Assessing the tactical environment forms part of the Intelligence Preparation of the Battlefield (IPB) process. IPB is a systematic, nonstop process of analysing the threat and environment in a geographic area and is designed to support staff estimates and military decision making. The IPB process may be divided into four steps, namely battlefield evaluation, TE, threat integration and production of a DSS overlay (Roux and Van Vuuren, 2007). The goals of IPB include the understanding of the effects that environmental conditions may have on units (enemy and friendly) operating within the Area of Responsibility (AOR) and the intelligent use of this information to improve combat power on the battlefield.

The terrain in the tactical environment may also have a considerable influence on reconnaissance and threat acquisition. Many sensor systems, such as Infra-Red (IR), Electro-optical (E-O) and optical sensors, require an optical or electronic Line-of-Sight (LOS) to detect a

threat. LOS refers to the existence of an established visible path (without the interference of terrain) between two objects. Sensors are therefore typically deployed in an array (sensor grid) such that the majority of the AOR is in the LOS of at least one sensor. Similarly, most weapon systems are constrained by the surrounding terrain to engage only those aerial threats within LOS and are therefore ineffectual against aircraft flying behind obstacles. The deployment of weapon systems at positions in an area where obstructions are minimal is therefore essential.

Terrain also has an influence on the ability of aircraft to approach, find and engage DAs. An experienced pilot may tactically manoeuvre his/her aircraft towards a DA, using the terrain to provide him/her with cover against GBAD weapon systems. The identification of techniques and paths that the OPposing FORce (OPFOR) may possibly employ within the tactical environment may lead to improved situational awareness and a means for better weapon system positioning.

Poor weather conditions degrade battlefield operations, affect the effectiveness of sensors and weapon systems, and influences the performance of personnel. Important elements of weather include atmospheric pressure, cloud cover, dew point temperature, humidity, precipitation, temperature, visibility, wind speed and wind direction. Severe weather conditions may cause it to be impossible to reach or deploy at certain positions, depending on the type of weapon system. Strong winds may have a significant influence on the range capability of a weapon system, while particularly strong cross winds may steer a projectile off course. Weather conditions may also have a significant impact on personnel and equipment. The influence of extreme temperatures have a more immediate effect on the human body than on electronic and mechanical systems. The soldier's first response to such extreme conditions is survival, which may affect his/her concentration and sound judgment.

7.2.5 Detection and identification of threats

During an attack, a set of actions is initiated in response (Potgieter, 2007). The initial defensive response is threat detection and identification. Any aerial vehicle, with the potential to cause damage to one or more of the DAs, is considered a threat. The threats are then prioritised by the TE system according to the level of danger they pose to the DAs. A number of engagements by the GBAD weapon systems are then suggested by the WA system in an attempt to prevent a successful attack on the DAs.

Threats are detected and monitored by a network of local and/or remote sensors called a *sensor grid*. The observed sensor data is then relayed to a Track Management (TM) system where it is formed into a *system track*, containing aircraft attributes, for each of the detected aircraft. The TM system is also responsible for Type Classification/Identification (TCI) and Hostility Classification/Identification (HCI) of the observed aircraft and for labelling the corresponding system tracks accordingly. The TCI process distinguishes between platform types such as *fixed wing, rotary wing, cargo, unmanned aerial vehicle, missile, EW platform*

and *unknown*. The HCI process may classify system tracks as either *friendly, hostile* or *unknown*.

Those system tracks that were classified as hostile or unknown serve as real-time input to the TE system. The TE system evaluates the behaviour of observed enemy aircraft in terms of the threat they may pose to DAs, using the system tracks and preliminary data. Each of the observed aircraft is assigned an estimated *threat value* with respect to each DA as an indication of the level of threat it poses to that DA, and prioritised accordingly. Threats are typically analysed according to a combination of their capability and intent (Roux and Van Vuuren, 2007). A more detailed explanation will be given in Section 7.2.9.

The simplest TE models are concerned only with sudden changes in aircraft behaviour and *flag* an operator if any or a selected subset of the aircraft attributes significantly deviates from their current values (Potgieter, 2007).

The output is stored in a two-dimensional matrix containing an estimated threat value for each threat-DA pair. Additionally, a TE model may also compile a list containing an estimated threat value for each threat in terms of the set of SAs as a whole. These are respectively called a *threat matrix* and a *combined threat list*. All the threats within the AOR should be included in each combined threat list – each threat only once – and these combined threat lists should be updated as regularly as the threat matrices are updated.

Given a combined threat list, the WA system solves an assignment problem, assigning available weapon systems to threatening aircraft within the AOR. Weapon systems are allocated to threatening aircraft according to a pre-specified objective as dictated by a WA *mode*. Weapon systems in an AD deployment may be allocated either with the aim to destroy as many of the OPFOR's aircraft as possible or with the aim to engage (scare) as many of the OPFOR's aircraft as possible. The former is known as *attrition* mode and the latter as *deterrence* mode. In attrition mode, a weapon system will typically engage an aircraft only once the aircraft is in the area where the weapon system is the most effective with respect to that aircraft. In deterrence mode, a weapon system will typically engage an aircraft as soon as possible. Weapon systems are only alerted when the operator sends an engagement order. The WA system must at all times be aware of the status of the weapon systems and DAs to ensure the validity of proposed assignments, and is therefore responsible for keeping track of the status of and operations involving these entities.

7.2.6 The Fire Control Operator

Due to the highly dynamic nature of modern aircraft, the number of events and short time spans of air strikes, defence forces are forced to make extensive use of DSSs for fire control (Le Roux *et al.*, 2006). Fire control is in essence the "optimal" assignment of weapon systems to threats (targets). These systems are collectively referred to as TEWA subsystems. A TEWA subsystem evaluates and prioritises possible threats, proposes the assignment of

resources to counter threats and schedules future assignments.

The Fire Control Operator (FCO) is responsible for fire control and uses a TEWA system for decision support (Potgieter, 2007). Fire control refers to the management of the weapon systems within an AD deployment, including assignment of weapon systems to threats and communications with weapon system operators. The proposed assignment list is provided to the operator by the WA system, via the Human Machine Interface (HMI). The operator may accept these assignments, but is also authorised to override the proposed pairs; imposing weapon system-threat pairs of his/her own choice instead. The consented assignments, together with expected hit/kill probabilities of the related engagements and other associated information, are stored in a so-called *active engagement list*.

The FCO may perform a number of functions to manipulate the TEWA input or output. The FCO is entitled to alter the ranking of a threat with respect to the other threats or set the priority of a specific threat to the maximum so as to invoke *enhanced reaction* by the TEWA system. Enhanced reaction refers to a predetermined *quick* response procedure to counter threats that are detected for the first time very close to the DAs (*i.e.* detected very late). The FCO also has the authority to create manual engagement, override TEWA proposed engagements and remove a manual or TEWA proposed engagement before the engagement orders are sent to the weapon systems. A subset of or all of the engagements may be suspended or cancelled by the FCO. In exceptional cases the FCO may change the properties of system tracks, including the hostility classification (friendly, unknown or hostile), the platform type (fixed wing, rotary wing, cargo, UAV, missile, or unknown) and the raid size, and may also remove a system track in its totality.

7.2.7 The flight path prediction module

A flight path prediction module forms part of the TEWA system and predicts the flight paths of threats for a number of future time steps (Potgieter, 2007). Both the TE and WA systems use such information to perform their respective computations. Some TE models compare flight path projections with predetermined profiles in an attempt to isolate possible attack techniques.

The WA system also depends on a flight path prediction module. Once an engagement order is sent to a weapon system, the threat is not engaged immediately and there is a reaction time involved in which the threat may change position. Some WA models may consider engagements for a number of future time steps and therefore require predicted future positions for all the threats in the AOR. The further into the future a flight path is predicted, the poorer that prediction is.

A flight prediction model may be used to predict the positions of enemy aircraft at future time steps. Given the position of a threat, there exists an optimal orientation for each weapon system along which to engage the threat. If an enemy aircraft is predicted to

be within range of a weapon system at a future time step, it is possible to determine an engagement efficiency probability for the weapon system in relation to the threat at that time step by simply reading out the entry in the Weapon System Efficiency Matrix (WSEM) corresponding to the position of the threat. If a threat is predicted to be outside the range of a weapon system at a future time step, the engagement efficiency probability for that weapon system with respect to the threat at that time step is taken as zero (Potgieter, 2007).

The most basic model for flight path prediction makes use of the observed kinematic data of the aircraft and assumes it will proceed at its current velocity (*i.e.* if an aircraft is flying in a straight line at 250 m/s at time step τ_0, it is assumed that the aircraft will continue flying in a straight line at 250 m/s at time step $\tau_0 + T_0$).

A second, more complex model is based on the assumption that the aircraft must be facing the asset when delivering a weapon. Note that this is not always necessary, since advances in weapon guidance systems enable aircraft to launch weapons at a standoff-distance and/or do not require the aircraft to be facing the related asset when the weapon is delivered. This conservative assumption assumes that the aircraft will in effect following the shortest possible path towards the asset.

A more reliable prediction may be made if the aircraft is expected to follow one of a number of known attack techniques. Given the position of the aircraft at a specific time step, probabilities may be associated with the attack techniques, indicating to what degree the aircraft is expected to follow each of the attack techniques. The predicted position of the aircraft at the next time step may be projected along the most probable attack technique or taken as an average of the positions projected along the respective attack techniques, weighted with their associated probabilities. A stochastic approach may be computationally more expensive, but may give a more realistic prediction of the effectiveness of weapon systems against enemy aircraft at future time steps and may absorb some deficiencies including sensor measurement errors, extrapolation errors, and variations in aircraft dynamics.

During weapon system deployment, the aim is typically to deploy weapon systems to protect the DAs "optimally". This is usually achieved by placing weapon systems such that the fire arc of each weapon system partially overlaps with those of adjacent weapon systems. A threat, being in the AOR at a given time, will thus only be within the fire arc of a few weapon systems, and the remaining weapon systems will have an efficiency of 0% with respect to this threat.

7.2.8 Command and Control

Advances in threat technology, the increasing difficulty and diversity of scenarios, and the volume of data and information to be processed under time-critical conditions pose major challenges to tactical Command and Control (C2), in particular TEWA (Paradis *et al.*,

2005). The dynamic environment in which these activities are conducted is one of high risk and high stress as it includes organised, intelligent and lethal threats. It is also uncertain due to the imprecise and incomplete nature of sensor data and intelligence, which places unpredictable demands on operators.

A C2 system is defined as the facilities, equipment, communications, procedures and personnel essential to a commander for planning, directing and controlling operations. *Command* may be defined as an authoritative act of making decisions and ordering action, and *control* as an act of monitoring and influencing this action. This definition emphasises the synergy required between human, doctrinal and Information Technology (IT) elements to ensure effective C2.

C2 in early warfare tended to be exercised by a single commander, who carried out all the planning, direction and monitoring functions, with assistance from a few aides and messengers only. During the 20^{th} century, this approach evolved to the concept of a C2 system. The military domain of C2 has for years been solving similar resource allocation problems applied to the weapon-target allocation problem (Naidoo, 2008). The use of TEWA is fundamental to any military command and control system, and is a specific case of the more general resource allocation problem.

The key enabler for C2 success is regarded as information superiority and the achievement of this, which has been an ongoing quest by military tacticians, is required to support decision makers in making sound, timely and informed decisions. Dynamic C2 and battle management functions require fast and effective decision aids to provide optimal allocation of resources for effective engagement and real-time battle damage assessment (Rosenberger *et al.*, 2005). Related information is very scarce and the inner workings of such a system are typically kept secret (Roux and Van Vuuren, 2007). As a result, not much research has been published on GBADS specific TEWA systems, especially in South Africa.

A military C2 system should provide the commander with a clear understanding of the developing operation so that forces can be directed in a manner to meet a specified objective. Such a system surveys the operational environment, assesses what actions to take (*i.e.* aiding the TE process), and uses available resources to implement those actions (*i.e.* aiding the WA process).

Some forms of C2 are primarily procedural or technical in nature, such as the control of air traffic and air space, the coordination of supporting arms, or the FC of a weapon system. Others deal with the overall conduct of military actions and involve formulating concepts, deploying forces, allocating resources and supervising troops. The scope of operation of a TEWA system spans the scope set by both these forms of C2, and therefore a thorough understanding of the applicable operational processes of C2 is required before an efficient TEWA system may be designed.

The C2 decision cycle may be described by the most basic, well known and widely accepted model of C2, namely Boyd's Observe-Orient-Decide-Act (OODA) loop. The OODA loop

model can be represented graphically as a circular connection of the four phases of the decision cycle, as shown in Figure 7.1. During the *observation* phase information is gathered, relevant to the decision at hand. Information may be collected from either internal sources (typically a feedback loop within the decision making entity), or from external sources (typically sensors or information sources outside the decision making entity).

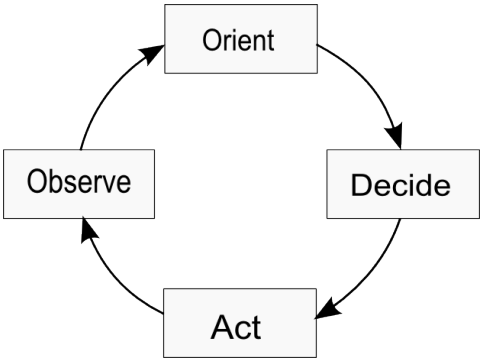


Figure 7.1: The OODA loop

Most of the cognitive effort during the decision process resides in the *orient* phase, which consists of two sub-phases: *destruction* and *creation*. A decision making entity will attempt to *destruct* or decompose a problem until the sub-problems are close to situations for which the decision maker has a plan. Problems are matched with their respective emergency plans, which are then combined or *created* into an overall plan of action.

If a decision maker creates a single feasible plan (during the orientation phase), the *decision* phase is simply whether or not to execute. The *action* node in Figure 7.1 represents the execution of a chosen course of action or plan. These actions may include a physical attack or movement, the issuance of an order, or a focus of effort on the sensor for a better observation during the next cycle of the process.

Tactical C2 can be divided into the following high-level functions (Leenen and Hakl, 2009):

1. Target detection - mainly through sensors;
2. Target tracking - use data fusion techniques;
3. Target identification - to determine true targets;
4. Threat evaluation - establish the intent and capability of potential threats;
5. Weapons assignment - target engagement.

7.2.9 Threat Evaluation and Weapon Assigment

In a GBADS context, where there are a number of prioritised DAs to protect, a TEWA system is applied in an *area-defence* role (Roux and Van Vuuren, 2007). A TEWA system

is thus at the core of a GBADS, providing customisable decision support to the FCO when prioritising threats and assigning weapons in a rapidly changing, high risk and highly stressful operational environment. A typical tactical situation in a GBADS environment exhibits all the trademarks of a complex system. Any model of the situation consists of a large, connected number of formative elements such as enemy aircraft, DAs, weapon systems and sensor systems. There is continual, dynamic, typically non-linear interaction between these elements such as looping and feedback of information. The system is open (even in the case of a large threat environment), far from equilibrium (often on the edge of chaos) and has history (*i.e.* the direction of time is important).

An operator is typically required to evaluate the tactical situation in real-time and protect DAs against enemy threats by assigning available weapon systems to engage enemy craft (Roux and Van Vuuren, 2007). This environment requires rapid operational planning and decision making under severe stress conditions, and the associated responsibilities are usually divided between a number of operators and computerised DSSs that aid these operators during the decision making processes.

TE establishes the intent and the capability of the potential threats within a certain Volume of Interest (VOI) and for a specific reference point (Benaskeur *et al.*, 2007; Paradis *et al.*, 2005). It refers to the ongoing process of determining if an entity intends to inflict evil, injury, or damage to the defending forces using specified rules to estimate the relative priority according to the level of threat posed (Naidoo, 2008).

The evaluation of targets as *threats* relies heavily on the use of the established situational picture and the available contextual information which may range from the locations of the DAs, attributes of platform types and attack techniques, weapon systems and surveillance systems, doctrine, intelligence reports, features of the terrain and the tactical environment, through to knowledge of the opposing force's structure and the recent history of its behaviours within the tactical environment (Roux and Van Vuuren, 2007). A TE model provides output in the form of a threat matrix.

Real-time TE is very challenging, mainly due to time constraints associated with the highly dynamic nature of the modern battlefield and the short time spans and intervals of strikes executed by modern weapon platforms (Project TEWA). Although traditionally ill-defined due to its cognitive nature, recent advances in real-time TE provide for the estimation of threat using mathematical models depending on two main threat attributes, namely capability and intent (Paradis *et al.*, 2005). *Capability* refers to the ability of a target to inflict damage to one or more DAs. Attributes considered when evaluating capability of a hostile aircraft or group of aircraft include size of a flight formation, proximity to the DAs, aircraft types and likely weapon and delivery types. *Intent* is the measure of the willingness or determination of a hostile aircraft to attack a DA. This is less trivial to estimate than capability, since it exists in the cognitive domain and cannot be measured directly by known factors in real-time.

WA, or weapon allocation, or weapon-target assignment (WTA), is a resource management process during which weapons are allocated to engage threats in the threat ranking list (Leenen and Hakl, 2009). The problem stated very simplistically is the assigning of n weapons to m targets (Naidoo, 2008). As with the resource allocation problem, the WA problem is also NP-complete and as such no exact methods exist for the solution of even relatively small sized problems. WA decisions are considered more easily quantifiable than TE, and thus the challenge lies more in the solution methodologies of the problem, rather than the formulation, as is the case with TE.

A WA system is required to provide the FCO with a *proposed assignment list* containing weapon system-threat pairings, that supports him/her when executing fire control by assigning weapon systems to observed threats. A WA system takes as input threat information, obtained from the TE system. The assignment problem is solved, given the current status of DAs, weapon systems status and capabilities (which includes terrain and weather) and the relevant threat information, resulting in a number of different weapon system-threat pairings. Threat information obtained from the TE system is required to be in the form of combined threat lists before the assignment process may commence. When the FCO authorises the assignment of a weapon system to a threat, the relevant weapon system must be informed. The output of the WA model framework is proposed assignment lists at the end of each time step.

It is assumed that at least unidirectional communication is possible from the command node to the weapon systems and that the command node has situational awareness of targets in the vicinity of the entire AD system (Le Roux *et al.*, 2006). WA considers time of opportunity, cost, effectiveness, availability and fire rank. There are six observable factors on which the probability of assignment is dependent. These factors are:

CPA Quality: The quality of the Closest Point of Approach (CPA) is a percentage to indicate if a target is directly approaching the weapon considered or if it will be a crossing target. This value is mapped to an interval state between 0% (passing tangential) and 100% (directly approaching) with a resolution of 5%.

Time of opportunity: The time a target is likely to spend in the launch envelope of the weapon expressed as a percentage of the maximum possible. Mapped similar to CPA.

Cost: The cost of the engagement. Each weapon system has an associated cost relative to the other weapon systems that may be considered.

Effectiveness: The effectiveness of the weapon system is dependent on the flight profile of the target that it is being considered for. It relates to an effectiveness index for the considered weapon system relative to others, with regard to the particular flight profile of the considered target.

Available: The availability of the weapon system. It can be either of two states, either true or false, to take into account whether the weapon system being considered is available for an assignment or has already been assigned.

Firing rank: The rank of the weapon-target pair. A rank is assigned based on whether the target is predicted to pass through the primary or secondary fire arc of the weapon system, through the launch envelope, but not through either of the fire arcs (low) or not through the launch envelope at all (zero).

Potgieter (2007) sort the targets from highest to lowest threat (rank) and considers them for WA in that order. The probabilities for all possible weapon pairs, with the considered target, are then sorted from high to low according to the assignment probability being true, *i.e.* the value on the "true" state of "Assignment". The pair with the highest probability is committed. If a weapon system had already been assigned to a target of a higher threat the next best pair is committed. A planned assignment is committed to be an assignment order when the target in that pair reaches a pre-determined threshold. All probabilities are recalculated in each time interval for all targets not already engaged.

It would obviously be advantageous to have a system that adapts as the operator becomes more proficient at evaluating threats and assigning resources. The system should then in principle improve as the operator gains experience. The WA extension is initially trained with a data set generated by the existing TEWA subsystem in automatic assignment mode. Once trained, the existing TEWA subsystem and extension are used in parallel - both outputs are presented to the operator. As the operator manually commits, accepts and rejects weapon-target pairs, the network is adapted on-line with the new cases.

TE is critically dependent on human cognition to make the call if an observed threat is indeed friend or foe, but with WA there is room for learning algorithms; algorithms put in place to assist the FCO in makings his/her decisions. In the next chapter we will see how Reinforcement Learning (RL) can be effectively applied in this extremely stressful and dangerous environment.

7.2.10 Applications

The literature showed that the methods used in formulating and solving the WA problem may be applied to fields as diverse as advertising in the contemporary business world, media allocation for an advertising campaign (Çetin and Esen, 2006) and so may also be applied to, amongst others, the scheduling of emergency rescue resources (Naidoo, 2008).

Chapter 8

The Weapon Assignment Application

8.1 Introduction

In the first few chapters of this dissertation we introduced different methods of Reinforcement Learning (RL) and illustrated some of its features. In this chapter we show how RL can be used to efficiently assign weapons in a simplification of the classical Weapon Assignment (WA) problem that was introduced in Chapter 7.

We assume a fixed world map, a fixed Defended Asset (DA) and four missiles at fixed positions. A threat is detected on the world map and flies towards the DA in a reasonably straight line. Our aim is to shoot it down as quickly as possible before it hits (and thus destroys) the DA.

8.2 Modelling the problem

We start off by assuming that the world map is a 7×7 grid. In later sections we use larger grids, but for now, we start small. As stated in the previous chapter, the object of the WA problem is to effectively assign weapons in order to defend a certain DA. The DA we want to defend is situated at the centre of the grid and is fixed. We use four missiles to defend the DA and once these missiles are placed, they are also fixed. We start off by considering symmetrically placed missiles, each two cells away from the DA: top, right, bottom and left. This grid is illustrated in Figure (8.1).

The threat flies in from one of the border cells and then flies radially towards the DA. Our aim is to eliminate the threat before it reaches the DA. Every weapon gets a turn to shoot

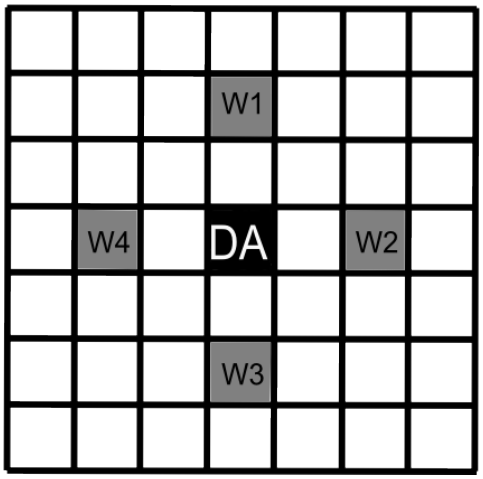


Figure 8.1: 7×7 world map

at the threat while it is in a specific cell, according to some strategy (from here on we will refer to this as the *policy*). When weapon i shoots and misses, it is the next weapon's turn, and so on. If the threat is eliminated, we have succeeded in our aim. If all four weapons miss, the threat moves one cell closer to the DA and again the weapons take turns to shoot at the threat. If the threat reaches the DA, it eliminates the DA. If the threat is on any of the weapons' positions, no shots can be fired for fear of injuring, or killing, the soldiers manning the missiles. In this case, the threat moves one cell closer to the DA, just as it would if all the weapons shot and missed. A typical flight pattern (assuming that all the weapons miss in each round) that we wish to avoid is illustrated in Figure 8.2.

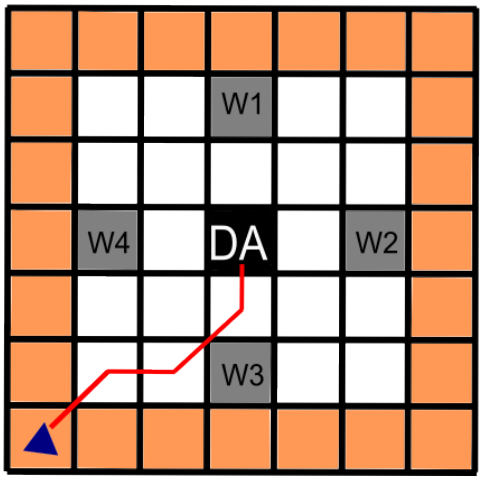


Figure 8.2: Flight pattern of threat eliminating DA

The rest of this section is divided into three parts corresponding to the important components that need to be modelled, namely states and actions, rewards and penalties, and the

policy.

8.2.1 States and actions

The world map is illustrated as a grid in order to exactly pinpoint different positions. The finer the grid, the more accurate the position of the threat can be calculated. In the terminology of RL, this also aids us in defining states. In this case, the weapon master bases his decisions on the position of the threat, the *position* of the threat being the *state*. We are working with a 7×7 grid, so there are $49 - 4 - 1 = 44$ (total states minus four weapons and one DA) different states that the threat can be in at any given time step within the episode where we have to take action. This leads to the question of how to define an episode. Thinking back to the blackjack example of Section 4.3.1, we recall that an episode starts the moment the first card is dealt and ends when someone either wins, loses or draws. This example works the same, but instead of someone winning, losing or drawing, someone eliminates, or is eliminated. In other words, the moment the threat is detected on the grid, an episode starts. That episode ends when either the threat is eliminated, or when the threat reaches the DA. The actions are the shooting order and in this case it is a numbered sequence, depicting which weapon shoots when, for example $\{3, 2, 1, 4\}$. If we have an $X \times X$ grid, there are $X^2 - 5$ states and with four weapons there are $4! = 24$ different actions (shooting orders).

8.2.2 Rewards and penalties

The WA problem is formulated as an undiscounted ($\gamma = 1$), episodic finite Markov Decision Process (MDP). It is *episodic* because the agent-environment breaks naturally into episodes– a threat flies in and is either shot down or shoots down the DA. It is *finite* because it is an MDP with finite state and action sets– there are $X^2 - 5$ states and $4! = 24$ actions. According to Sutton and Barto (1998*a*), it is appropriate to discount for continuing tasks and not for episodic tasks. Thus future rewards are taken into account more strongly and the agent becomes more farsighted.

A reward of $+3$ is given when the threat is eliminated by the first weapon in the shooting order. If the second, third or fourth weapon eliminates the threat a reward of 0 is given. If all four weapons miss, a reward of -1 is given and if the DA is eliminated, a reward of -5. This encourages the system to eliminate the threat as quickly as possible. These different rewards are summarised in Table 8-I.

It is worth noting that in this dissertation we only check that the correct weapon fires first. We do not check whether the correct weapon fires second, third, or fourth. It would be relatively easy to incorporate this into our problem, but we chose not to focus on that.

The probability of a hit is inversely proportional to the distance. This is not controlled by the agent, therefore it is part of the environment. We reward the first weapon for eliminating

Case	Reward
Weapon 1 eliminates threat	+3
Weapon 2 eliminates threat	+0
Weapon 3 eliminates threat	+0
Weapon 4 eliminates threat	+0
All four weapons miss	−1
DA is reached	−5

Table 8-I: Obtaining the rewards

the threat, because we want to eliminate the threat as soon as possible. If the second, third or fourth weapon eliminate the threat we do not penalise these weapons, because it is not "wrong", it is just not exactly what we wanted. By rewarding only the first weapon we encourage the algorithm to use the weapon with the highest chance of eliminating the threat. If all four weapons miss, we penalise that particular state-action pair by giving it a reward of -1. When the DA is reached, a reward of -5 is given. Better results were obtained when the absolute value of the penalty for eliminating the DA was greater than the reward for eliminating the threat. More on that later in the chapter.

8.2.3 The policy

This then brings us to the policy, which is a function of the state-action pair (s, a). Our initial policy is to assume the weapons shoot in numbered ordering, from 1 to 4. This is not a good policy, seeing that it does not take into account where the threat is located at that particular time. We wish to iterate and improve this policy until we have one that agrees with our intuition, which is that the weapon closest to the threat should fire (that weapon also has the highest probability to eliminate the threat). For this simple problem the solution is known and this will be used to evaluate the RL algorithm in the following chapter.

8.3 Solving the problem

The problem introduced in the previous section is solved with two different algorithms. The first algorithm is Monte Carlo ES, which is a Monte Carlo control algorithm assuming exploring starts (the reader is referred to Algorithm 5 in Chapter 4.5). The second algorithm is Q-learning, which is an off-policy Temporal Difference (TD) method (the reader is referred to Algorithm 10 in Chapter 5.7).

The problem is kept the same for both algorithms as far as possible in order to compare their results. First, an episode of WA is simulated by randomly choosing a starting cell for the threat. From the previous section the threat flies in from the border, and that is still the case, but in order to encourage exploration we decided to rather choose a starting

cell randomly. That way we ensure that all states are chosen as starting positions. This position is the current state of the threat. The distance from the threat to each of the four weapons is then calculated with Pythagoras (as seen in Figure (8.3)), and based on that, a kill probability P_{kill} is assigned to each weapon using a lookup table.

The rest of the section illustrates how the kill probabilities are determined and how they are used, as well as how variable rewards and firing distances are used.

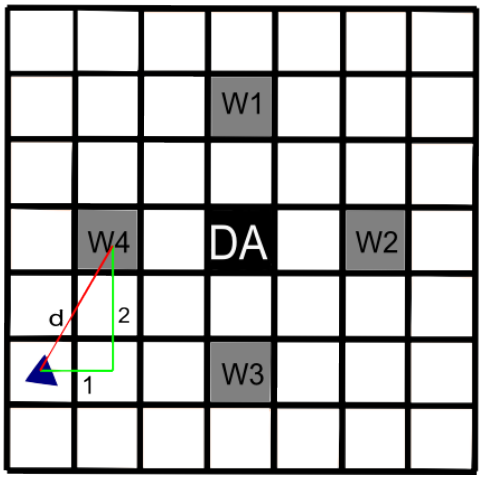


Figure 8.3: Calculating minimum distances

8.3.1 Determining the kill probabilities

The calculation of the kill probabilities explained in this subsection is part of the formulation of the WA problem, and not of the RL algorithm.

The firing distance of a weapon can be thought of as a circle around the weapon. If the threat is within this circle, it will be shot down according to some probability. If the threat is in the circle and close to the weapon, then it has a high probability of being shot down than if it was on the edge of the circle.

For this reason we divide the firing circle into smaller concentric circles. We give each concentric circle a different probability– a higher probability closer to the weapon and a small probability on the edge of the circle.

If the firing distance of the weapon is between one and two, as in Figure (8.4), we do not divide the circle into smaller circles. The threat can only move one cell at a time, and therefore it does not make sense to divide this circle. Table 8-II shows how P_{kill} is calculated with a firing distance in the interval $[1, 2)$, where d is the distance from the threat to the weapon and f the firing distance. Note that $GR = \frac{1+\sqrt{5}}{2}$ is the *golden ratio* (Livio, 2003).

The "Golden Ratio" has no significance in the context of the problem other than having utility as a scaling factor to generate interesting P_{kill} values that fall within the range observed in simulation (using detailed models of intercept) and that reported from field trials. In future, if a proper data set can be acquired for P_{kill}, the scaling factor may be replaced by a validated statistical distribution to generate the data.

Figure 8.4: Firing distance in the interval $[1, 2)$

Distance	P_{kill}
$0 < d < f$	$0.5GR \approx 0.81$
$d = 0$ or $d \geq f$	0

Table 8-II: P_{kill} lookup table for $[1, 2)$

If the firing distance is between two and three, we divide the firing circle into two concentric circles as shown in Figure (8.5). Table 8-III shows how P_{kill} is calculated with a firing distance in the interval $[2, 3)$.

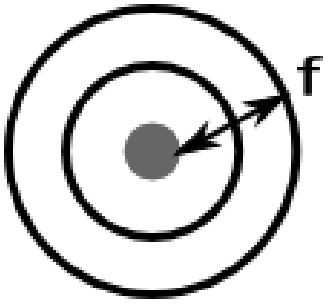


Figure 8.5: Firing distance in the interval $[2, 3)$

Distance	P_{kill}
$0 < d < f/2$	$0.5GR \approx 0.81$
$f/2 \leq d < f$	$0.4GR \approx 0.65$
$d = 0$ or $d \geq f$	0

Table 8-III: P_{kill} lookup table for $[2, 3)$

If the firing distance is between three and four, we divide the firing circle into three concentric circles as shown in Figure (8.6). Table 8-IV shows how P_{kill} is calculated with a firing distance in the interval $[3, 4)$.

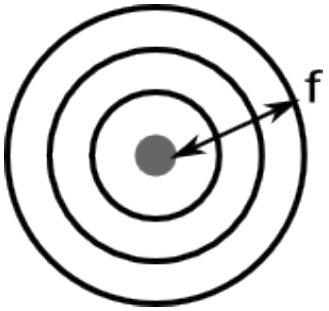


Figure 8.6: Firing distance in the interval $[3, 4)$

Distance	P_{kill}
$0 < d < f/3$	$0.5GR \approx 0.81$
$f/3 \leq d < 2 \times f/3$	$0.4GR \approx 0.65$
$2 \times f/3 \leq d < f$	$0.3GR \approx 0.49$
$d = 0$ or $d \geq f$	0

Table 8-IV: P_{kill} lookup table for $[3, 4)$

If the firing distance is between four and five, we divide the firing circle into four concentric circles as shown in Figure (8.7). Table 8-V shows how P_{kill} is calculated with a firing distance in the interval $[4, 5)$.

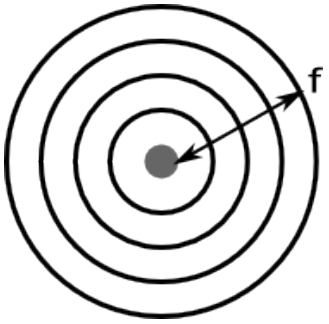


Figure 8.7: Firing distance in the interval $[4, 5)$

Distance	P_{kill}
$0 < d < f/4$	$0.5GR \approx 0.81$
$f/4 \leq d < 2 \times f/4$	$0.4GR \approx 0.65$
$2 \times f/4 \leq d < 3 \times f/4$	$0.3GR \approx 0.49$
$3 \times f/4 \leq d < f$	$0.2GR \approx 0.32$
$d = 0$ or $d \geq f$	0

Table 8-V: P_{kill} lookup table for $[4, 5)$

When the firing distance is greater than five, we divide the firing circle into ten concentric circles, no matter how large the firing distance is. Figure (8.8) shows these ten concentric circles and Table 8-VI shows how we computed P_{kill} for the interval $[5, \infty)$.

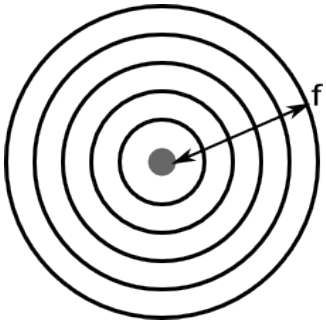


Figure 8.8: Firing distance in the interval $[5, \infty)$

Distance	P_{kill}
$0 < d < f/10$	$0.50GR \approx 0.81$
$f/10 \leq d < 2 \times f/10$	$0.45GR \approx 0.73$
$2 \times f/10 \leq d < 3 \times f/10$	$0.40GR \approx 0.65$
$3 \times f/10 \leq d < 4 \times f/10$	$0.35GR \approx 0.57$
$4 \times f/10 \leq d < 5 \times f/10$	$0.30GR \approx 0.49$
$5 \times f/10 \leq d < 6 \times f/10$	$0.25GR \approx 0.40$
$6 \times f/10 \leq d < 7 \times f/10$	$0.20GR \approx 0.32$
$7 \times f/10 \leq d < 8 \times f/10$	$0.15GR \approx 0.24$
$8 \times f/10 \leq d < 9 \times f/10$	$0.10GR \approx 0.16$
$9 \times f/10 \leq d < 10 \times f/10$	$0.05GR \approx 0.08$
$d = 0$ or $d \geq f$	0

Table 8-VI: P_{kill} lookup table for $[5, \infty)$

8.3.2 Using the kill probabilities

Suppose the firing distance is $f = 2.5$. That means we use the case shown in Table 8-III. If the distance between the threat and a weapon is less than 1.25 units, the chance that the threat will be shot down is 81%. If the distance between the threat and a weapon is at

least 1.25, but less than 2.5, the chance that the threat will be shot down is 65%. If the threat is on the same position as the weapon (hence the distance between them being zero) or outside of the firing circle, the threat will not be shot down (0% chance).

8.3.3 Variable reward

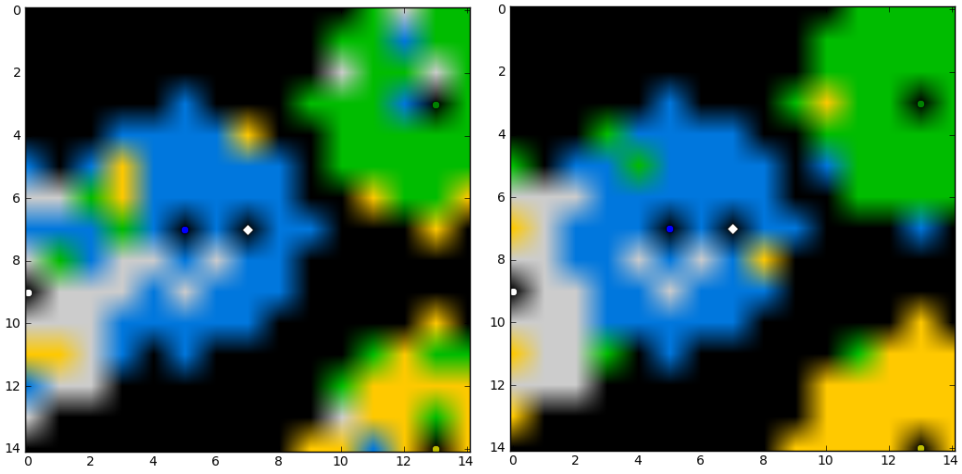
In Section 8.2.2 we mentioned that better results are obtained when the absolute value of the penalty for eliminating the DA is greater than the reward for eliminating the threat. The following 15×15 grids illustrate this. We chose a 15×15 grid, because a 7×7 grid does not really get the point across, and performing thousands of simulations on a 71×71 grid takes a long time.

We evaluate each state 50 times giving a total of 11,000 simulations with the Monte Carlo algorithm. Figure (8.9) shows how the same simulations differ after 1,100 simulations when given different rewards and Figure 8.10 after 11,000 simulations. The black areas are those that are outside any of the weapons' firing ranges. The coloured areas correspond to the different weapons. The blue area corresponds to the blue weapon (weapon 1), the green areas to the green weapon (weapon 2), and so forth.

In both figures, the rewards were given as shown by Table 8-VII.

	Rewards			
Case	Scheme 1 Figure (a)	Scheme 2 Figure (b)	Scheme 3 Figure (c)	Scheme 4 Figure (d)
Weapon 1 eliminates threat	+12	+12	+12	+12
Weapon 2 eliminates threat	+9	+0	+0	+9
Weapon 3 eliminates threat	+6	+0	+0	+6
Weapon 4 eliminates threat	+3	+0	+0	+3
All four weapons miss	−1	−1	−4	−4
DA is reached	−10	−10	−15	−15

Table 8-VII: Obtaining the rewards



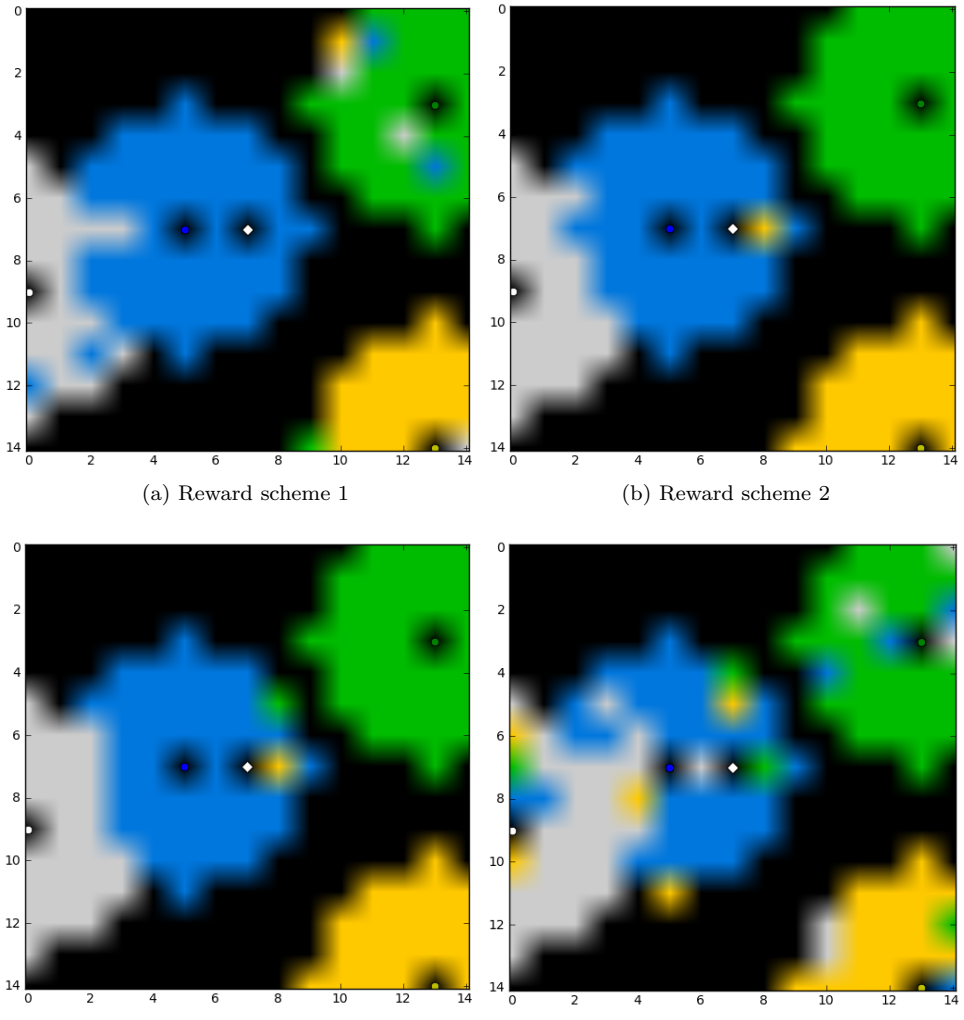
(a) Reward scheme 1

(b) Reward scheme 2

(c) Reward scheme 3

(d) Reward scheme 4

Figure 8.9: Different reward schemes after 1,100 simulations



(a) Reward scheme 1

(b) Reward scheme 2

(c) Reward scheme 3

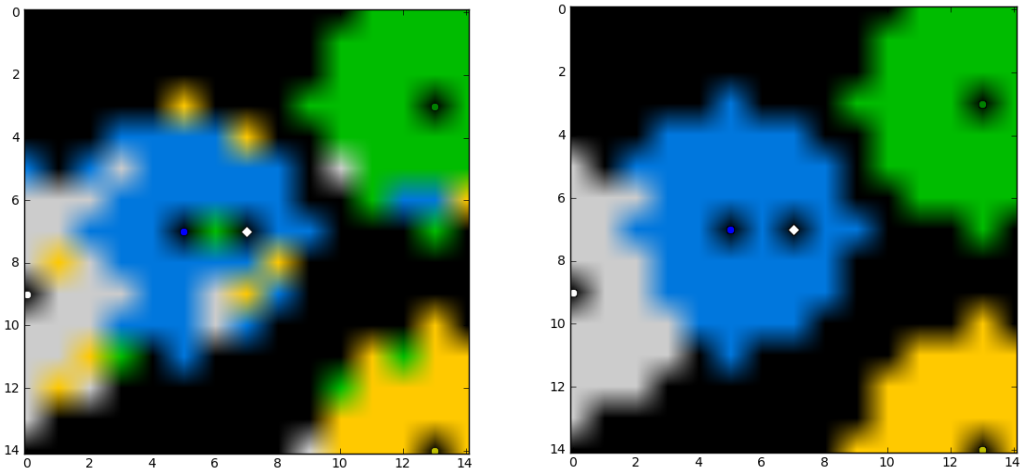
(d) Reward scheme 4

Figure 8.10: Different reward schemes after 11,000 simulations

From the different reward schemes we see that the rewards need not be very large or very small numbers, what matters is the size of the rewards in relation to each other. To illustrate this we added another reward scheme with smaller numbers. The rewards are shown in Table 8-VIII and the results are illustrated in Figure (8.11).

Case	Rewards
Weapon 1 eliminates threat	+3
Weapon 2 eliminates threat	+0
Weapon 3 eliminates threat	+0
Weapon 4 eliminates threat	+0
All four weapons miss	−1
DA is reached	−5

Table 8-VIII: Obtaining the rewards



(a) After 1,100 simulations (b) After 11,000 simulations

Figure 8.11: The reward scheme for Table 8-VIII

Table 8-IX shows the number of errors per figure after 11,000 simulations as well as the error percentages for each of the figures in Figure (8.10) and Figure (8.11)(b) which we refer to as "Figure (e)" in Table 8-IX. These errors are counted just as we counted the discrepancies in the previous examples. We compare the results with the known solution, that is, the weapon closest to the threat should fire first.

There are mainly two types of reward schemes here: those that give a reward no matter which weapon eliminates the threat and those that only give the first weapon in the shooting order a reward for eliminating the threat. The two reward schemes with the highest error percentages are Figure (a) and Figure (d). These two are also the only two that use the scheme that give rewards no matter which weapon eliminates the threat. It makes sense that it is better to only reward the first weapon in the shooting order for eliminating the threat.

Using the reward scheme that rewards only the first weapon in the shooting order for eliminating the threat, we obtain good results by penalising the system if all four the weapons failed to eliminate the threat and even better results when we combined this reward scheme with a large penalty for eliminating the DA. The best results are obtained when the absolute value of the penalty for eliminating the DA was greater than the reward given to the first weapon in the shooting order for eliminating the threat. It does not really matter how large the rewards are, what matters is how large they are in relation to each other. From this experiment it is evident that Figure (8.11) (Figure (e)) has the best reward scheme among those that were tested, and for the rest of this dissertation we use this reward scheme.

	Rewards				
Case	Figure (a)	Figure (b)	Figure (c)	Figure (d)	Figure (e)
Weapon 1 eliminates threat	+12	+12	+12	+12	+3
Weapon 2 eliminates threat	+9	+0	+0	+9	+0
Weapon 3 eliminates threat	+6	+0	+0	+6	+0
Weapon 4 eliminates threat	+3	+0	+0	+3	+0
All four weapons miss	−1	−1	−4	−4	−1
DA is reached	−10	−10	−15	−15	−5
Number of states	220	220	220	220	220
Number of errors	9	1	2	20	0
Error %	4.09%	0.45%	0.91%	9.09%	0%

Table 8-IX: Error percentages according to variable rewards

8.3.4 Another word on firing distances

In a 7×7 grid, the maximum distance between two points (in a straight line) occurs when one point is at $(0, 0)$ and the other at $(6, 6)$, or $(0, 6)$ and $(6, 0)$. This distance is equal to the hypotenuse if half of the grid is considered as a right angled triangle. This amounts to $r = \sqrt{X^2 + X^2} = \sqrt{2X^2} = \sqrt{2}X$, where X is the size of the grid, in this case 7. If the firing distances of the weapons, f, are assigned this value, $f = r$, it would mean that all of the weapons could shoot anywhere on the grid; there is no state where the threat is "safe".

The smaller the value of the firing distance f (small relative to X), the smaller the area covered by the weapons. A small value of f makes for aesthetically pleasing pictures, but only because there is very little, or no, overlap between the weapons. If is better for the weapons to overlap, because if weapon i shoots and misses, we know that at least one other weapon will be able to reach the target and, hopefully, shoot it down. With a small value of f there are no anomalies in the pictures and the result corresponds to the analytical solution, but with a large value of f, there are sometimes discrepancies concerning the allocation of weapons. For instance, weapon 1 might be closer to the threat than weapon 3, but the picture shows weapon 3 needs to shoot. One explanation could be that not enough simulations were run. Another explanation could be that the discrepancy is due to the fixed way we calculate the flight path.

When f is small, it also takes much longer for the simulation to execute. This might seem strange, but consider the following: we have a large grid, say 71×71, a small firing distance f, and randomly placed weapons. A state is chosen (assume for the moment that we are starting at $(0, 0)$), as well as an action. If the threat is not within the firing distance f from the weapon, the weapon will miss with a 100% chance. If all four weapons are out of this firing distance, the threat moves towards the DA, and keeps on doing so until it is within firing distance from a weapon. And even then there is not a 100% chance that the weapon *will* eliminate the threat, only an 81% chance at best (refer to Table 8-VI). So for a small f, the episodes are much longer than it would be with a larger f. A larger f could possibly result in the weapon eliminating the threat in the first round of the episode, whereas it would be impossible if it were outside of the firing distance.

8.4 Implementation of the MC algorithm

Before simulating any episodes, we have to initialise a few functions. The first one is the policy $\pi(s)$ which we initialise as $\pi(s) = \{1, 2, 3, 4\}$ for all states s as previously mentioned. Next we initialise the action-value function $Q(s, a) = 0$ for all state-action pairs (s, a). $Q(s, a)$ is an $X \times X \times 4!$ grid that can be thought of as X^2 file cabinets standing in an $X \times X$ formation, each having $4! = 24$ drawers. Each file cabinet corresponds to a state in the grid and each drawer corresponds to a specific shooting order. We also initialise an $X \times X \times 4! \times 2$ $Returns(s, a)$ matrix. This also corresponds to a file cabinet system, but each drawer now has two compartments. The first compartment is for keeping the sum of rewards and the second compartment keeps track of how many times we have visited that specific drawer (or shooting order). This is necessary for MC but not for Q-learning. For Q-learning we do not need a specific policy.

When an episode starts, a state-action pair is chosen. We choose a starting position and a random shooting order (remember that there are $4! = 24$ possibilities), which is also for exploratory purposes. To encourage exploration even further we decided to walk through the whole grid, starting at $(0, 0)$ and ending at $(X - 1, X - 1)$ instead of choosing a random starting position. That way we are sure that no state is skipped and the algorithm is forced to visit each state at least once. This enables us to see many different outcomes for the same state. For this simple problem we have a model– the known sequence of state visits.

Armed with this starting position and a random shooting order, the algorithm lets the weapons shoot at the threat according to the shooting order. If one of the weapons eliminate the target (preferably the first weapon), we know that this is possibly a good action to take when in that state. The $Returns(s, a)$ matrix's entry for that particular state-action pair is changed by incrementing it by the proper reward value. If, however, none of the weapons manages to eliminate the threat, we return to our original policy, namely shooting in the order for that particular state (which we look up in the policy matrix). If the threat is then eliminated, the entry in the $Returns(s, a)$ matrix for that particular state-action pair is

changed, otherwise the threat moves on and a new state is encountered. From here until the end of the episode we do not take any exploratory action, we simply look up the shooting order in the policy matrix. The exploratory random action is only taken at the start of an episode. We thus have two ways of exploring: by choosing a random action at the start of an episode and by traversing the whole grid a number of times as opposed to selecting a starting state at random.

We specify that the algorithm visits every state n times, thus the total number of simulations are $(X^2 - 5) \times n$, where n is a user defined number. To ensure that the algorithm does not miss any dependencies from previous states, we divided this process into five simulation blocks. The algorithm starts at state $(0, 0)$ and visits that state $\frac{n}{5}$ times, then moves on to the next state and visits that $\frac{n}{5}$ times. After having visited state $(X - 1, X - 1)$ $\frac{n}{5}$ times, the algorithm starts at state $(0, 0)$ and again visits that state $\frac{n}{5}$ times and so forth. Every state is visited $\frac{n}{5}$ times and the whole process is repeated five times.

When the threat is eliminated, the episode ends. If all four weapons miss, the threat moves one position towards the DA. The next position in the flight pattern is decided on as follows. Each cell in the grid is assigned a dummy value: the DA is assigned 0 and all eight neighbours of the DA is assigned 1, and so forth. The threat always flies to a cell with a lower number, hence moving closer to the DA.

8.4.1 Improving the policy

At the end of an episode, we need to improve the policy. We look at the current policy and decide where we can make improvements, what works and what not. The way we measure how "good" a policy is, is through the action-value function $Q(s, a)$. Every time a state-action pair is encountered, the corresponding entry in the $Returns(s, a)$ matrix is changed. This can be thought of as finding file cabinet s and locating its drawer a. We increment the number in the first compartment by the reward obtained during that visit and we increment the second compartment by one to say that we have visited that particular state-action pair. As soon as a state-action pair is visited, we flag that particular $Returns(s, a)$ cabinet. In this cabinet we look up the drawers whose values have changed and we average that value by dividing the first compartment's value by the second compartment's value. This averaged number goes into the corresponding drawer in the $Q(s, a)$ file cabinet. So at the end of the episode we have averaged all the visits to state-action pairs encountered in that episode. Also at the end of the episode we walk through the whole grid to find where the maximum of $Q(s, a)$ occurred each time. Thus, we look through each of the X^2 file cabinets (the $Q(s, a)$ file cabinets) and determine in which of the 24 drawers the maximum occurred. We are not interested in what that maximum is, but only which action (shooting order) led to it. This new shooting order then replaces the old one in the policy $\pi(s)$ matrix.

8.5 Implementation of the TD algorithm

Initialisation with the Q-learning algorithm is much simpler than with the MC method. All that needs to be initialised is the action-value function $Q(s, a) = 0$ for all state-action pairs (s, a). $Q(s, a)$ is an $X \times X \times 4!$ grid that can be thought of as X^2 file cabinets standing in an $X \times X$ formation, each having $4! = 24$ drawers, just as in the previous section. Each file cabinet corresponds to a state in the grid and each drawer corresponds to a specific shooting order. As previously mentioned, we do not need a specific policy for Q-learning.

When an episode starts, a state s is chosen. To encourage exploration we walk through the whole grid, starting at $(0, 0)$ and ending at $(X - 1, X - 1)$ instead of choosing a random starting position. That ensures that no state is skipped and the algorithm is forced to visit each state at least once. This enables us to see many different outcomes for the same state.

For each step of the episode, we choose an action a (shooting order) from the state s using the policy derived from $Q(s, a)$. In Sutton and Barto (1998a) they select this action by using an ϵ-greedy policy where $\epsilon \leq 1$, but we set $\epsilon = 0$, because we already have the condition built in that all state-action pairs are visited a number of times.

Armed with this starting position and shooting order, the algorithm lets the weapons shoot at the threat according to the shooting order. The rewards are assigned as follows: +3 if the first weapon in the shooting order eliminates the threat, +0 if any other weapon eliminates the threat, -1 if all four weapons miss and -5 if the DA is eliminated. If all four the weapons miss, we also observe the next state s'.

We update the state-action value at (s, a) by moving the difference between the old and new action-value a fraction of the way towards the old value:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]. \tag{8.1}$$

Again we specify that the algorithm visits every state n times, thus the total number of simulations are $(X^2 - 5) \times n$, where n is a user defined number.

When the threat is eliminated, the episode ends. If all four weapons miss, the threat moves one position towards the DA. The next position in the flight pattern is decided on exactly as in the MC implementation.

8.5.1 Shifting the parameters

In the MC implementation, $\gamma = 1$ was used as a rule throughout all the simulations. If $\gamma = 1$ in the Q-learning environment, the $Q-$values diverge (Watkins and Dayan, 1992). We affirmed this experimentally, but the experiment itself will not form part of this dissertation.

The learning rate

The learning rate parameter (α) limits how quickly learning can occur, thus to what extent the newly acquired information overrides the old information. In this specific algorithm, it directs how quickly the Q-values can change with each state-action change. A factor of 0 causes the agent not to learn anything, while a factor of 1 causes the agent to consider only the most recent information. If α is too small, learning happens very slowly. If α is too high, then the algorithm might not converge (Cline, 2004).

The discount factor

As we already know, the discount factor (γ) determines the importance of future rewards. A factor of 0 will make the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the Q values will diverge. If γ is low, immediate rewards are optimised, while higher values of γ cause the learning algorithm to count future rewards more strongly.

The greediness parameter

The purpose of following an ϵ-greedy policy is to explore all state-action pairs as the number of runs goes to infinity. We already built this condition into the simulation environment, thus rendering the need for more exploration redundant. In a greedy method, as we have here where $\epsilon = 0$, the algorithm might miss optimal solutions. However, if ϵ is too high it causes the agent to waste time exploring suboptimal state-action pairs that have already been visited. For the purpose of this study we chose to only look at the greedy case where $\epsilon = 0$ and vary the α and γ values. More on this in the next chapter.

8.5.2 Improving the policy

The action-value function $Q(s, a)$ that is learned, directly approximates $Q^*(s, a)$, the optimal action-value function, independent of the policy being followed. The policy determines which state-action pairs are visited and updated, however, all that is required is that all pairs continue to be updated.

8.6 Concluding remarks

In this chapter we applied MC with exploring starts as well as the offline TD algorithm, Q-learning, to the simplified WA problem. In the next chapter we see the results of these applications.

Chapter 9

Results

9.1 Introduction

In this chapter we implement the algorithms presented in the previous chapter, namely Monte Carlo (MC) with exploring starts (ES) and the Temporal-Difference (TD) learning algorithm, Q-learning. Our main focus is on how the results differ between these two algorithms. We start off by explaining how we obtained the error figures, after which we proceed with MC examples. Then we move on to TD examples and in the last section we discuss the results.

9.2 Obtaining the error figures

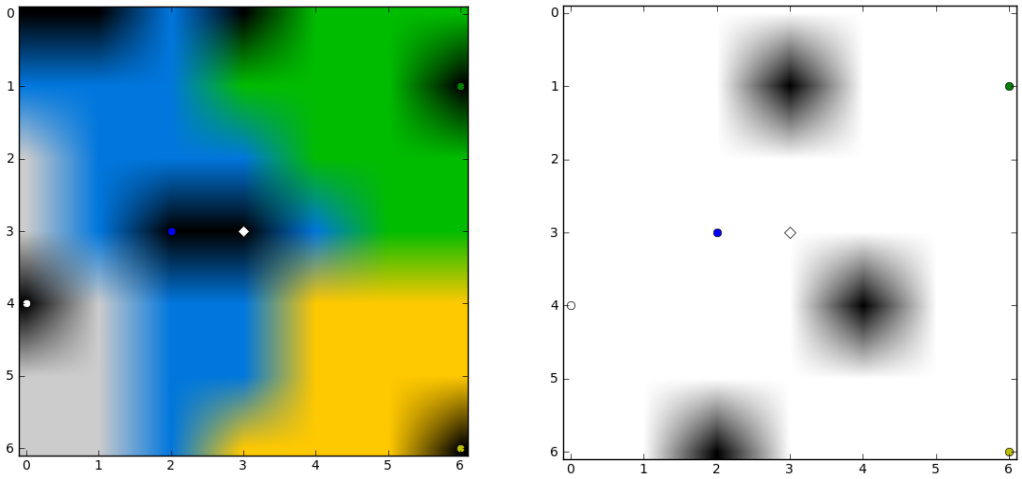
The analytical solution is that the weapon closest to the threat fires first. This solution is compared to the solution obtained during the simulations. At whichever state the simulations' grid differed from the firing grid, a black spot is plotted, thereby generating the error figure.

9.3 MC Examples

In this section we look at a few simulated examples. We first consider 7×7 grids with randomly placed weapons. We change the firing distances and play around with the number of simulations per state and then do the same for 71×71 grids.

9.3.1 The 7×7 grid

In Figure (9.1) we show a 7×7 grid with randomly placed weapons. Each state is simulated 550 times, thus giving us a total of 24,200 simulations. The firing distance is equal to 3 and the images indicate the first weapon in the shooting order. Figure (a) shows the result after 24,200 simulations and Figure (b) the errors.



(a) Results after 24,200 simulations

(b) Errors after 24,200 simulations

Figure 9.1: A 7×7 grid with randomly placed weapons

The four weapons are each marked with a coloured dot on the grid and the defended asset (DA) is shown as a white diamond in the centre. The blue area corresponds to that part of the grid covered by weapon 1 (blue dot). The green area corresponds to the part of the grid covered by weapon 2 (green dot), the yellow area to the part covered by weapon 3 (yellow dot) and the grey area to the part covered by weapon 4 (white dot). The black spots on the grid in Figure (a) are the states where no shots are fired. In the previous chapter we mentioned that no shots are fired if the threat is in the same position as one of the weapons, or if the threat is outside of the weapons' firing range. Also, if the threat is in the same position as the DA it does not help firing at it since it is already lost.

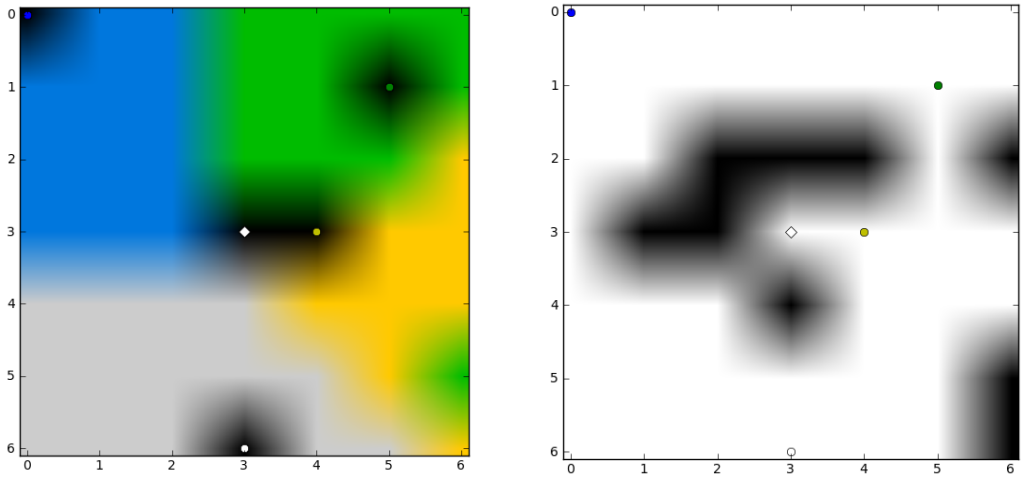
From Figure (9.1)(b) we see only three errors. The top two black spots are because weapon 1 should have fired, while the bottom black spot is because weapon 4 should have fired. In this particular example we have an error percentage of 6.122%. Note that these erronous actions taken are not completely "wrong", it is just not the best actions that could have been taken in that situation.

A possible reason for these discrepancies is the fact that, even though the "wrong" weapon shoots and eliminates the threat, its Q-value is probably still larger than other state-action pairs with negative Q-values. When the wrong weapon shoots and eliminates the threat,

that particular state-action pair receives a zero reward, whilst state-action pairs in which all the weapons miss, receive a negative reward. Thus, taking the maximum over all actions within a given state results in the algorithm choosing this wrong weapon to shoot first, because the corresponding state-action pair's Q-value is the largest (or possibly the only non-negative Q-value). The same goes for a correct weapon that is marked "wrong" by the algorithm– the maximum kill probability given to any weapon is 81%, so there is a chance of the correct weapon missing the threat. That particular Q-value could then be negative, and stay negative for a few iterations– long enough for the algorithm to assign a wrong weapon.

Next we look at results where the firing distance was 7, which is the length of the grid. Each state was visited 5,000 times. Again, the blue area corresponds to the part of the grid covered by weapon 1, the green area to weapon 2, the yellow area to weapon 3 and the grey area to weapon 4.

From Figure (9.2) we see quite a few errors, 9 in total, giving us an error percentage of 18.367%. When the firing distance is large compared to the grid length, overlapping occurs between the weapons. This leads to higher error percentages and discrepancies, like the green spot at the bottom of Figure (a) which should clearly have been a yellow spot. It is possible that the green spot is a state that was not yet visited by a successful weapon and is thus a remnant of a time when the original policy designated that cell as "green".



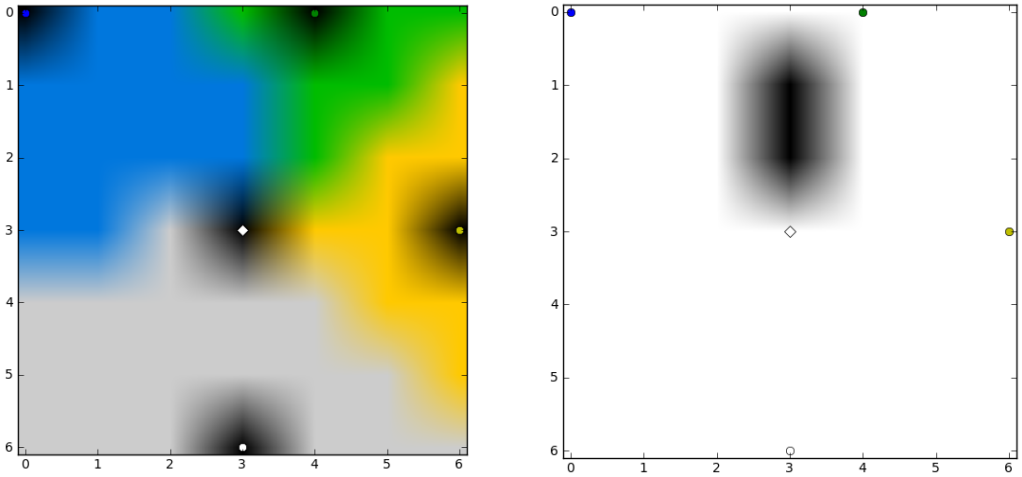
(a) Results after 220,000 simulations

(b) Errors after 220,000 simulations

Figure 9.2: A 7×7 grid with randomly placed weapons

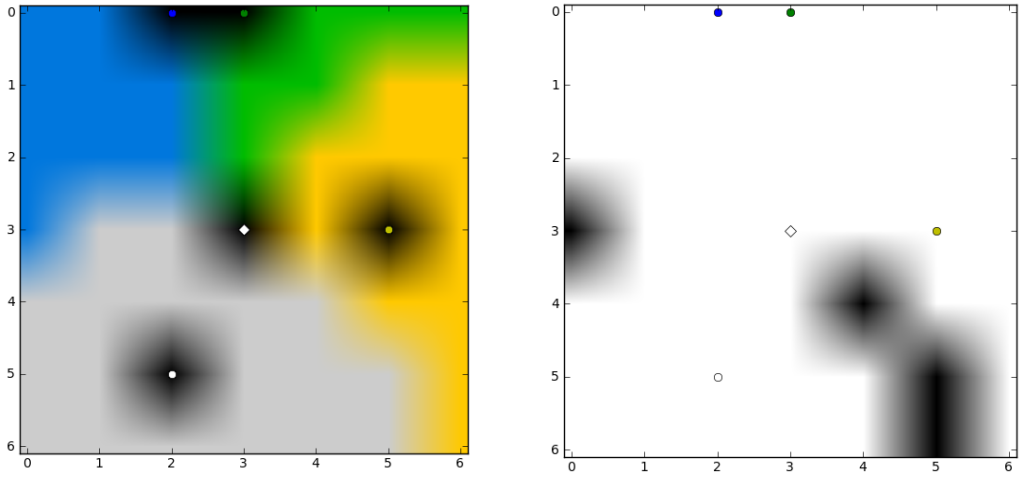
Two more sets of results for the 7×7 grid with firing distance 7 can be seen in Figure (9.3) and Figure (9.4). In Figure (9.3) the weapons are nicely positioned, so there should be minimal overlapping. As expected, fewer errors were made than in the previous example. In fact, only two errors were made, giving us an error percentage of 4.082%.

In Figure (9.4), four errors were made, giving us an error percentage of 8.163%.



(a) Results after 23,000 simulations (b) Errors after 23,000 simulations

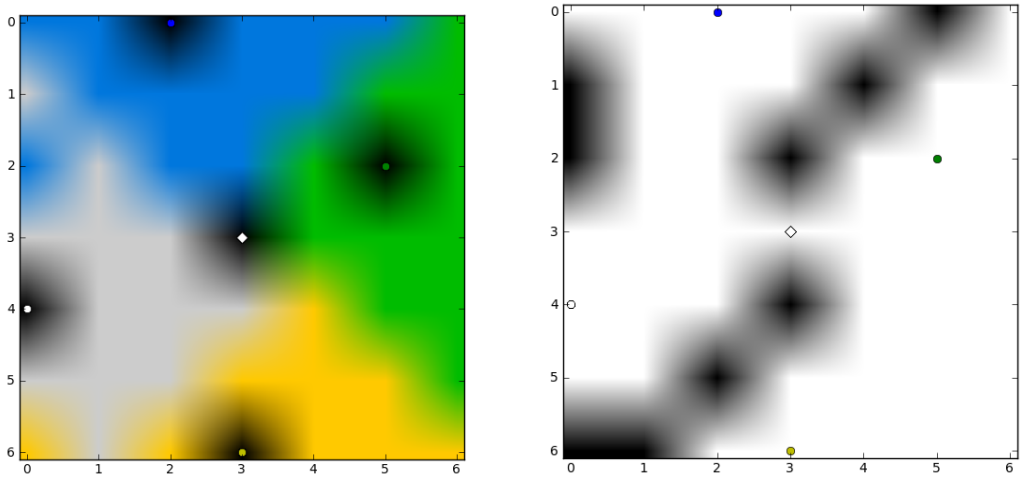
Figure 9.3: A 7×7 grid with firing distance 7



(a) Results after 23,000 simulations (b) Errors after 23,000 simulations

Figure 9.4: A 7×7 grid with firing distance 7

Figure (9.5) shows a random weapon layout where each weapon has maximum firing distance $\sqrt{2}X \approx 9.899$. Here the firing distance is even larger than in the previous examples, thus leading to greater overlapping. In this example, nine errors were made, leading to an error percentage of 18.367%.



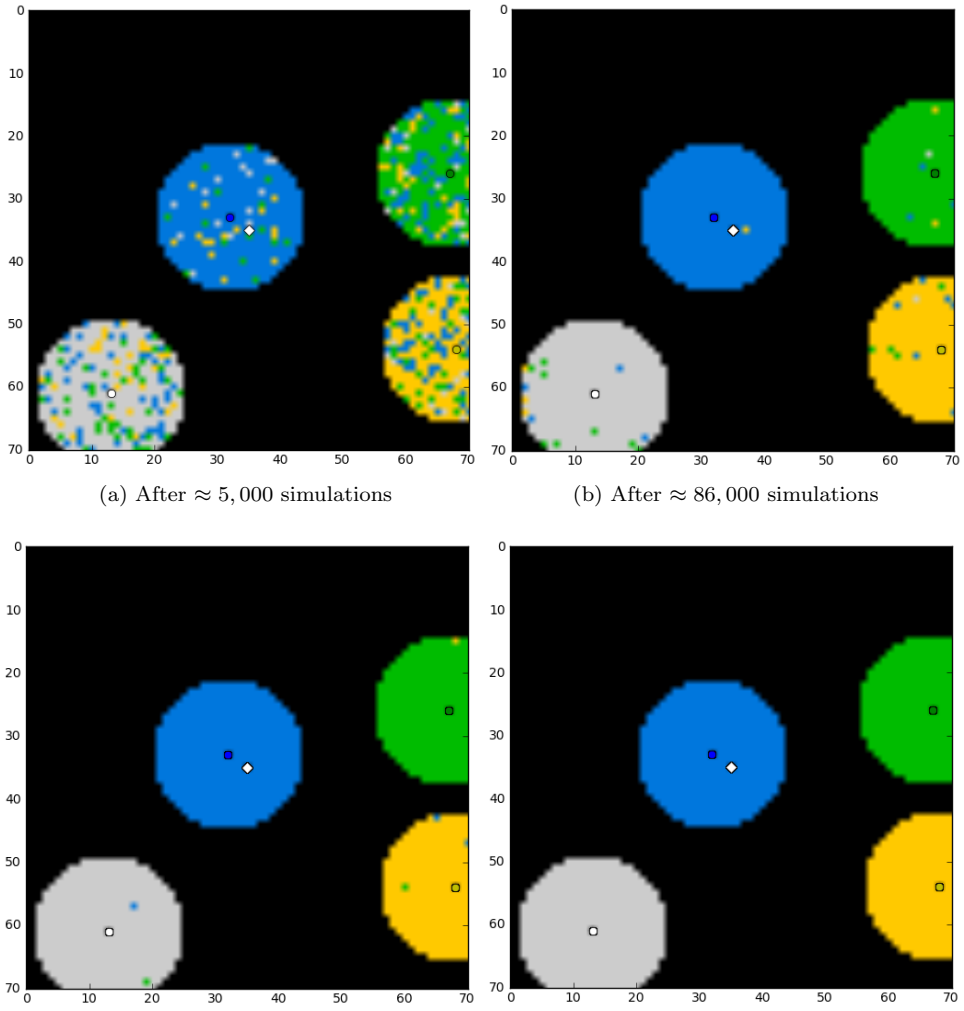
(a) Results after 23,000 simulations

(b) Errors after 23,000 simulations

Figure 9.5: A 7×7 grid with maximum firing distance

9.3.2 The 71×71 grid

In this subsection we show how the same methods and train of thought for the 7×7 grid can be applied to much larger grids. The spots of (different) colour seen inside the circles are due to the fact that that particular state was not visited by a successful weapon and is thus a remnant from a time when the original policy designated that state as being shot at by a different ("wrong") weapon.



(a) After $\approx 5,000$ simulations

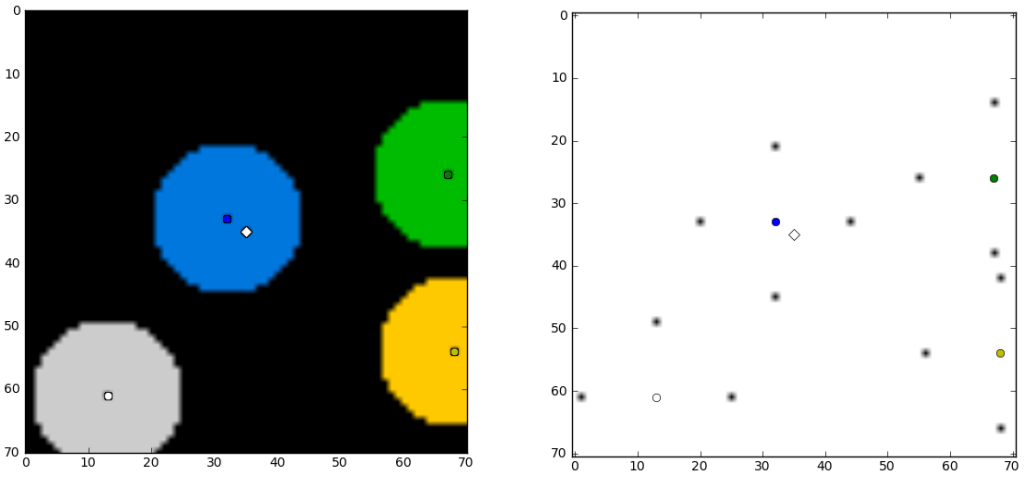
(b) After $\approx 86,000$ simulations

(c) After $\approx 166,000$ simulations

(d) After $\approx 252,000$ simulations

Figure 9.6: A 71×71 grid with 50 simulations per state and firing distance 12

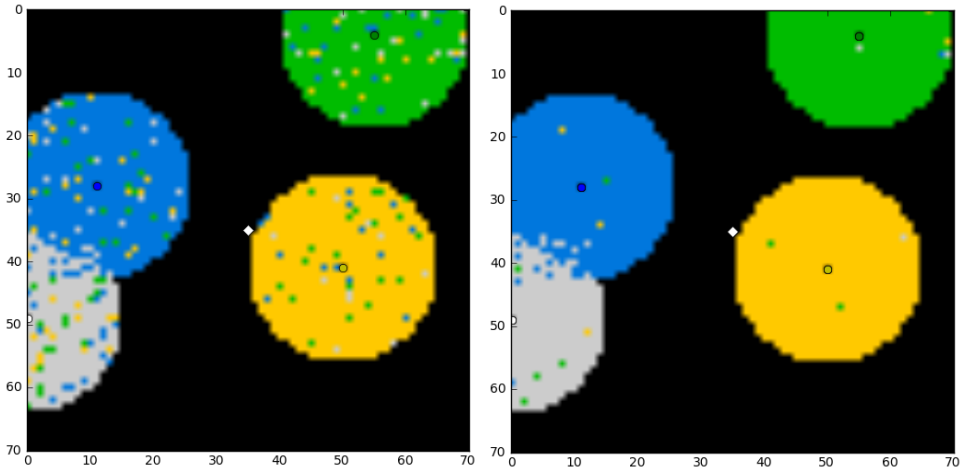
Figure (9.6) shows four images depicting the simulation results for a 71×71 grid with maximum firing distance equal to 12. Each state was simulated 50 times, thus producing approximately 252,000 simulations. Figure (9.7)(a) shows the final simulation result after approximately 252,000 simulations and Figure (b) the errors made. The error percentage for this example is 0.258%. It is interesting to note that the firing circles in Figure (a) are not completely round, they are slightly flattened at the sides, at the top and at the bottom. This is precisely where the errors are made. This could be a result of a rounding error or the way the problem was formulated, *i.e.* a situation where a "$<$" should have been a "$\leq$".



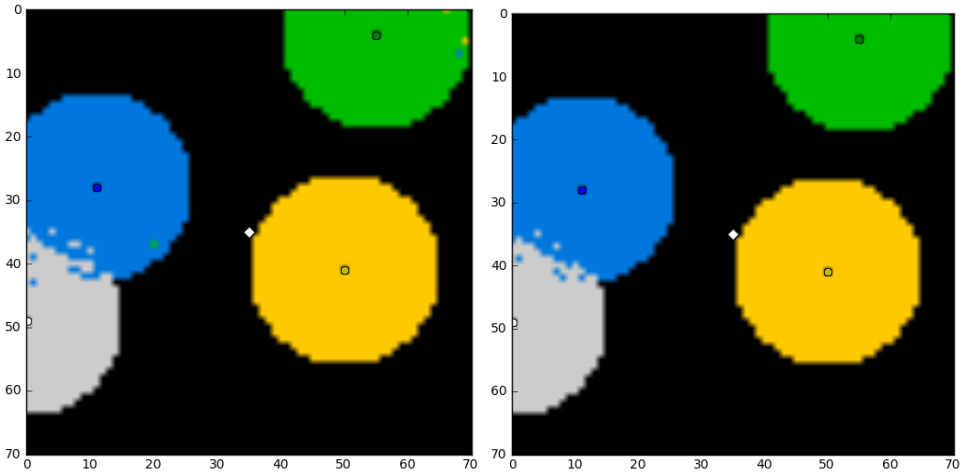
(a) Results after $\approx 252,000$ simulations

(b) Errors after $\approx 252,000$ simulations

Figure 9.7: A 71×71 grid with 50 simulations per state and firing distance 12



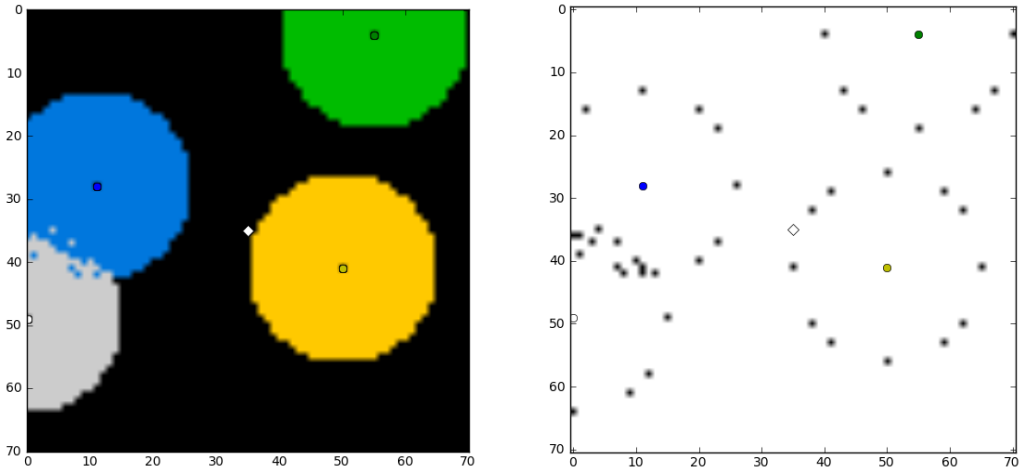
(a) After $\approx 90,000$ simulations (b) After $\approx 450,000$ simulations



(c) After $\approx 809,000$ simulations (d) After $\approx 1,300,000$ simulations

Figure 9.8: A 71×71 grid after 250 simulations per state and firing distance 15

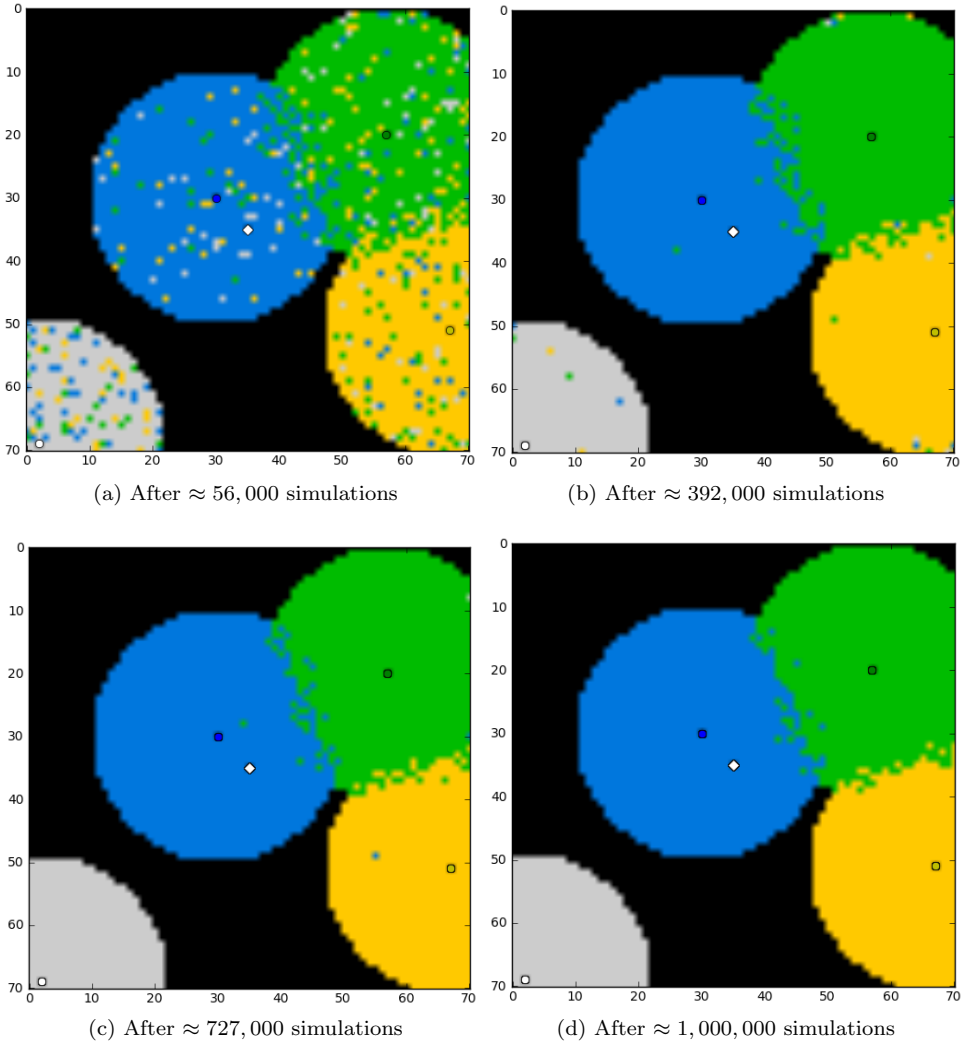
Figure (9.8) shows the results for a 71×71 grid with maximum firing distance of 15. Each state is visited 250 times, bringing the total to approximately 1,300,000 simulations. Figure (9.9)(b) shows the errors and the error percentage is 0.833%. Here we also see the slightly flattened firing circles, but even more so than in the previous example. The flattening is more evident in larger circles. Once again the errors made were on the edges of the firing circles, but more than in the previous example. There were also a few errors where weapons 1 and 4 overlap.



(a) Results after $\approx 1,300,000$ simulations

(b) Errors after $\approx 1,300,000$ simulations

Figure 9.9: A 71×71 grid with 250 simulations per state and firing distance 15



(a) After $\approx 56,000$ simulations

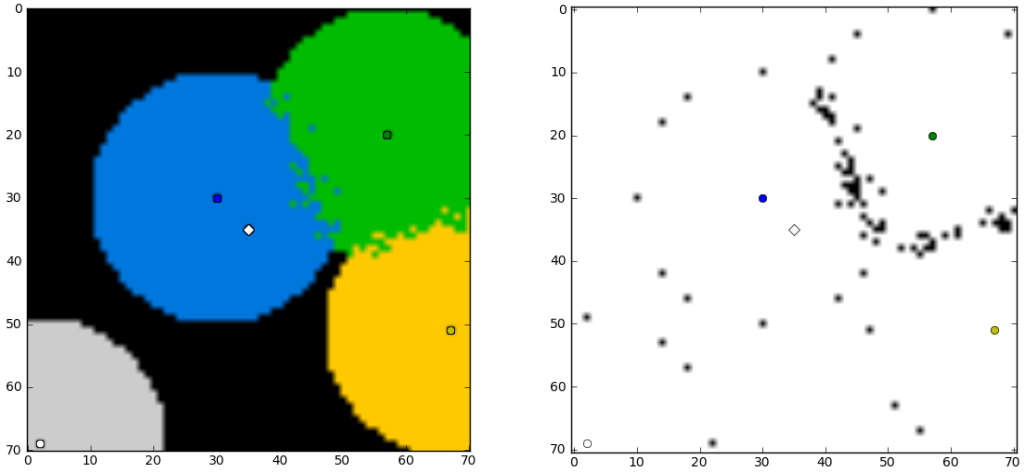
(b) After $\approx 392,000$ simulations

(c) After $\approx 727,000$ simulations

(d) After $\approx 1,000,000$ simulations

Figure 9.10: A 71×71 grid after 200 simulations per state and firing distance 20

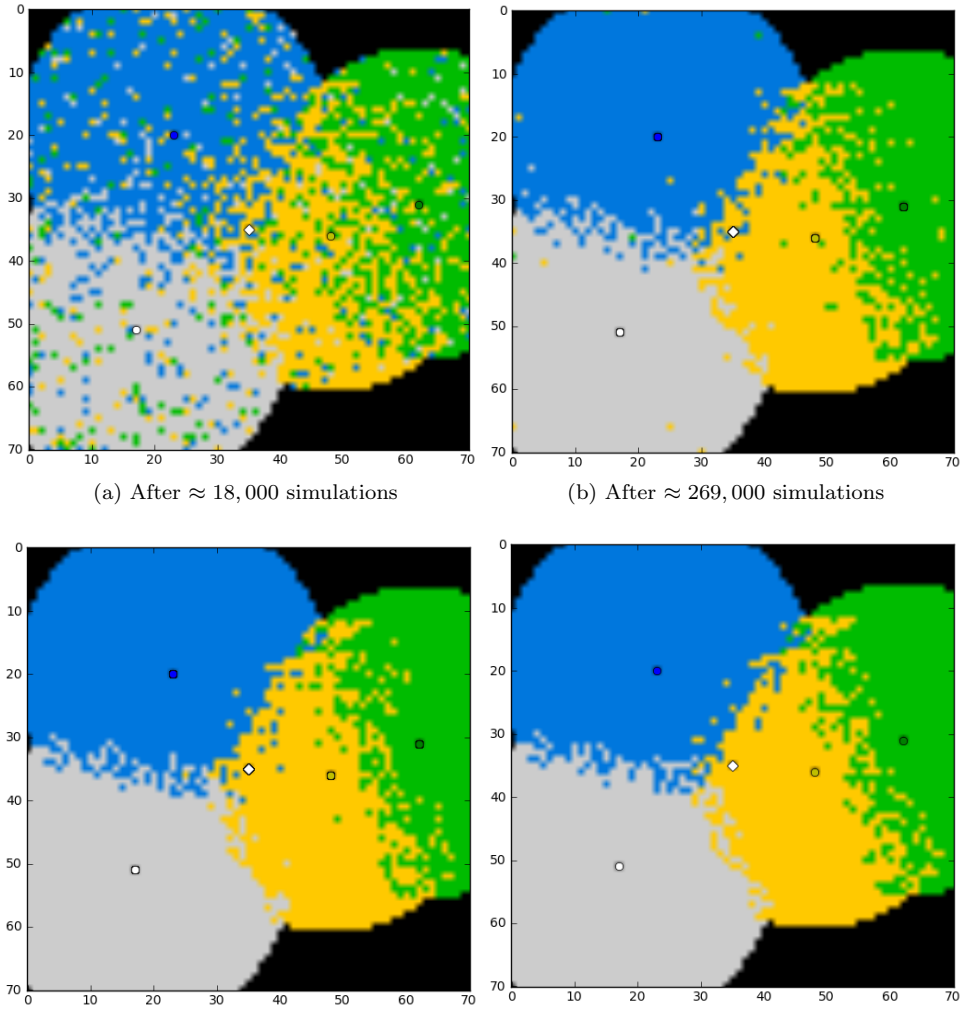
Figure (9.10) shows the simulation results for a 71×71 grid with maximum firing distance of 20. Each state was simulated 200 times, leading to a total of approximately 1,000,000 simulations. Figure (9.11)(b) shows the errors and the error percentage is 1.508%. The error percentage is larger than in the previous examples due to the larger firing distance as well as the fact that weapon 2 overlaps with both weapon 1 and weapon 3's firing circle.



(a) Results after $\approx 1,000,000$ simulations

(b) Errors after $\approx 1,000,000$ simulations

Figure 9.11: A 71×71 grid with 200 simulations per state and firing distance 20



(a) After $\approx 18,000$ simulations

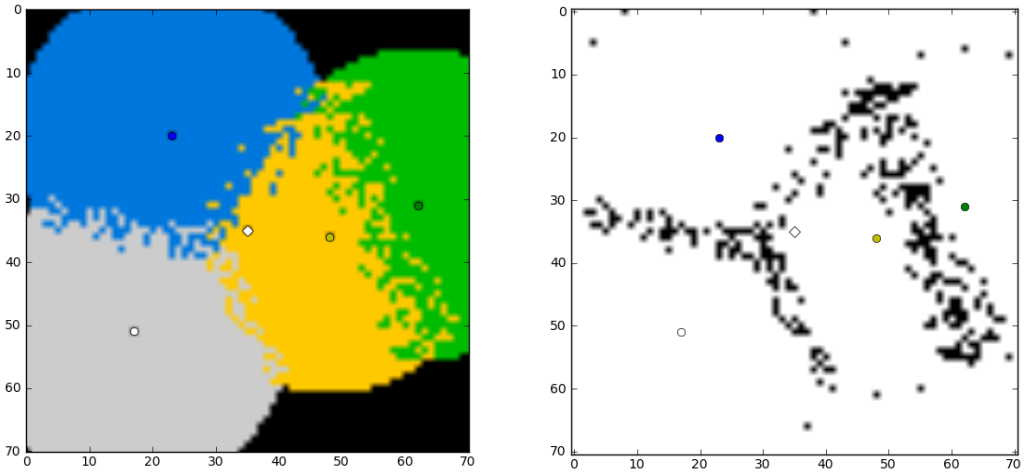
(b) After $\approx 269,000$ simulations

(c) After $\approx 537,000$ simulations

(d) After $\approx 806,000$ simulations

Figure 9.12: A 71×71 grid after 160 simulations per state and firing distance 25

Figure (9.12) shows a 71×71 grid with maximum firing distance of 25. Each state is simulated 160 times, leading to a total of approximately 806,000 simulations. Figure (9.13) shows the errors and the error percentage is 5.991%. The higher error percentage is due to the larger firing distance and the even greater overlapping of the weapons' firing circles.



(a) Results after $\approx 806,000$ simulations (b) Errors after $\approx 806,000$ simulations

Figure 9.13: A 71×71 grid with 160 simulations per state and firing distance 25

An action (shooting order) consists of four weapons and not just one. Suppose that we encounter a state where we expect weapon 3 to shoot. Further suppose that the greedy action we take at the start of the episode dictates that the weapons shoot in the order $\{1, 3, 2, 4\}$. Weapon 1 shoots and misses and then weapon 3 shoots and eliminates the threat. Weapon 1 obtains a reward of zero for missing the threat and weapon 3 obtains a reward of zero for being the second weapon to eliminate the threat. It is possible that, although this particular Q-value is zero, it is still larger than the other Q-values for that state. Thus, at the end of the episode this particular state-action pair's policy entry is changed to $\{1, 3, 2, 4\}$, although it should actually have been weapon 3 shooting first.

9.4 TD Examples

In this section we look at a few simulated examples. As in the previous section we change the firing distances and play around with the number of simulations per state. We only consider 71×71 grids.

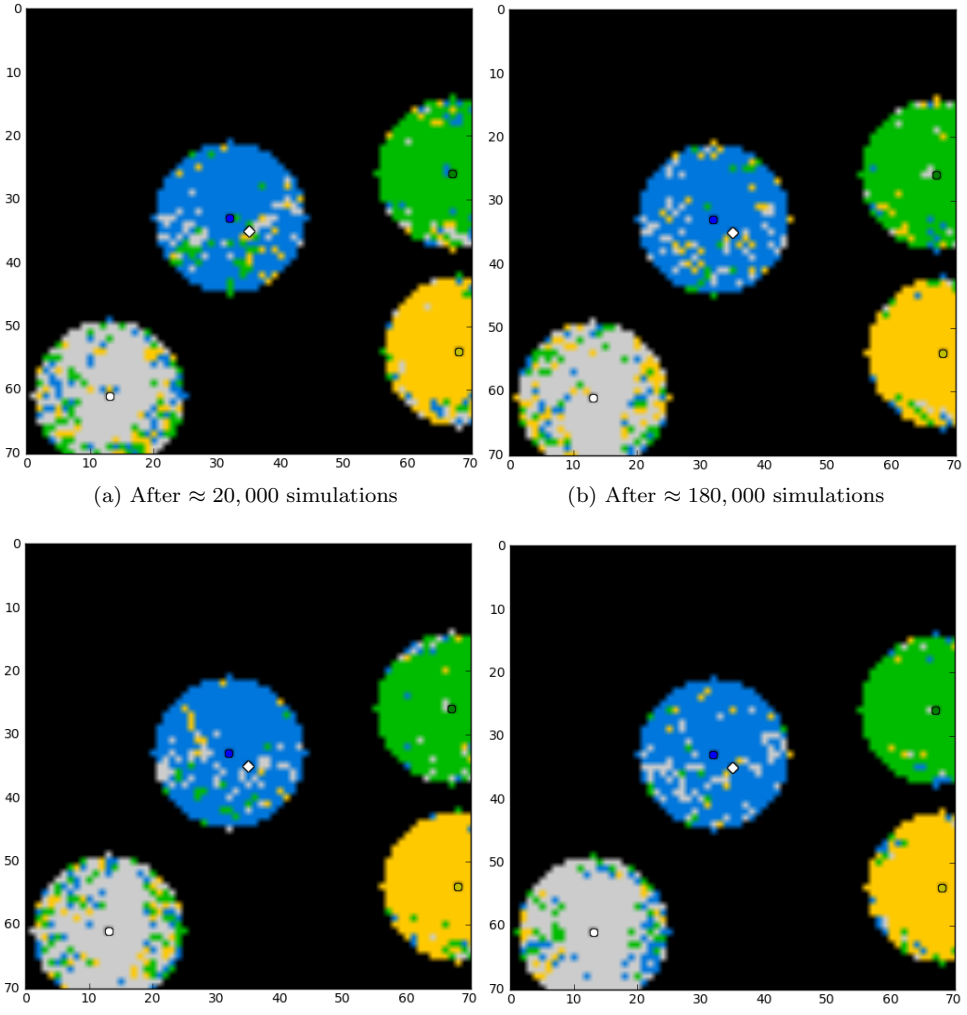
9.4.1 The 71×71 grid

As mentioned in the previous chapter, we cannot use $\gamma = 1$ as we did for the MC algorithm. We chose to keep $\epsilon = 0$ and vary the γ- and α-values. Through experimentation we saw that the best values for γ and α were in the ranges $0.2 \leq \gamma \leq 0.7$ and $0.1 \leq \alpha \leq 0.2$. Table 9-I gives a summary of our findings when applied to the same 71×71 grids as in the previous examples.

		Error % for various firing distances			
γ	α	12	15	20	25
0.2	0.1	4.939%	7.677%	11.287%	16.703%
0.3	0.1	4.999%	8.034%	11.645%	14.719%
0.4	0.1	**4.483%**	7.796%	11.168%	15.989%
0.5	0.1	4.701%	8.709%	**9.919%**	12.974%
0.6	0.1	6.784%	7.816%	10.137%	13.747%
0.7	0.1	5.297%	**6.903%**	10.950%	**11.744%**
0.2	0.2	9.601%	15.216%	18.231%	24.936%
0.3	0.2	9.502%	14.660%	16.425%	22.317%
0.4	0.2	7.499%	14.104%	15.314%	20.532%
0.5	0.2	8.411%	13.053%	16.128%	17.616%
0.6	0.2	7.935%	12.597%	14.779%	17.397%
0.7	0.2	8.272%	11.545%	14.204%	17.814%

Table 9-I: Error percentages

The minimum errors for the four firing distances occur when $0.4 \leq \gamma \leq 0.7$ and $\alpha = 0.1$. Using these parameter combinations, the following results are obtained. The same weapon layouts and firing distances are used as for the 71×71 grids in the MC section. Every state is simulated 500 times, bringing the total to approximately 500,000 simulations.



(a) After $\approx 20,000$ simulations

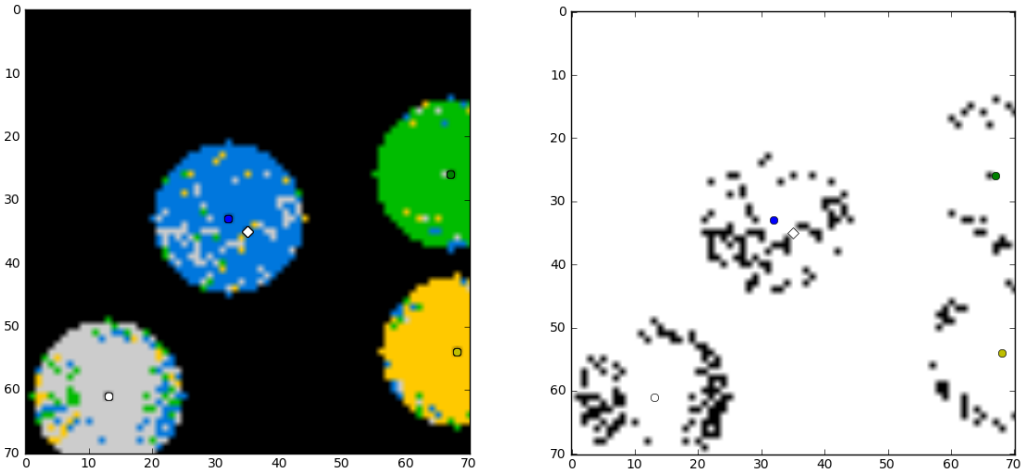
(b) After $\approx 180,000$ simulations

(c) After $\approx 340,000$ simulations

(d) After $\approx 500,000$ simulations

Figure 9.14: A 71×71 grid with 500 simulations per state and firing distance 12

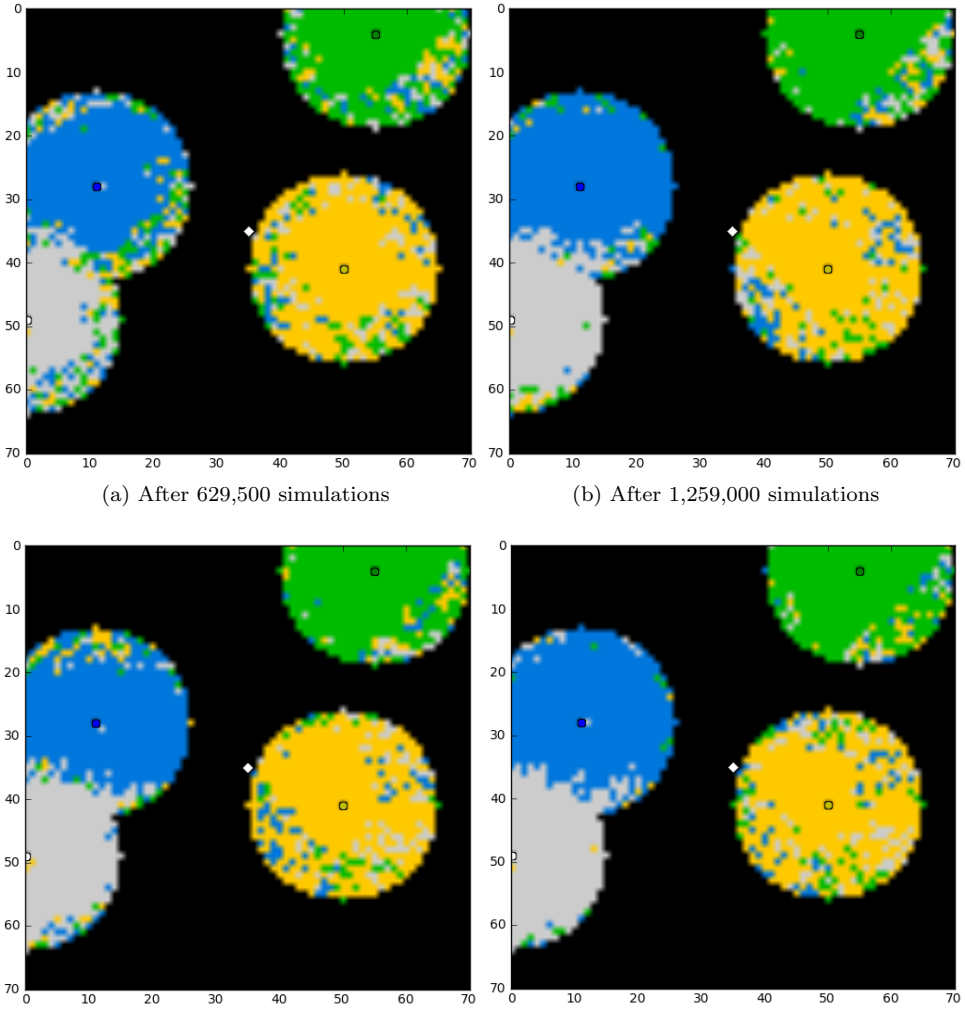
From Figures (9.14)(a)-(d) we see that learning for a 71×71 grid with firing distance 12 is very slow. This could be due to the small α-value (learning rate). Figure (9.15)(b) shows the errors made and the error percentage here is 4.483%.



(a) Results after 2,518,000 simulations

(b) Errors after 2,518,000 simulations

Figure 9.15: A 71×71 grid with 500 simulations per state and firing distance 12



(a) After 629,500 simulations

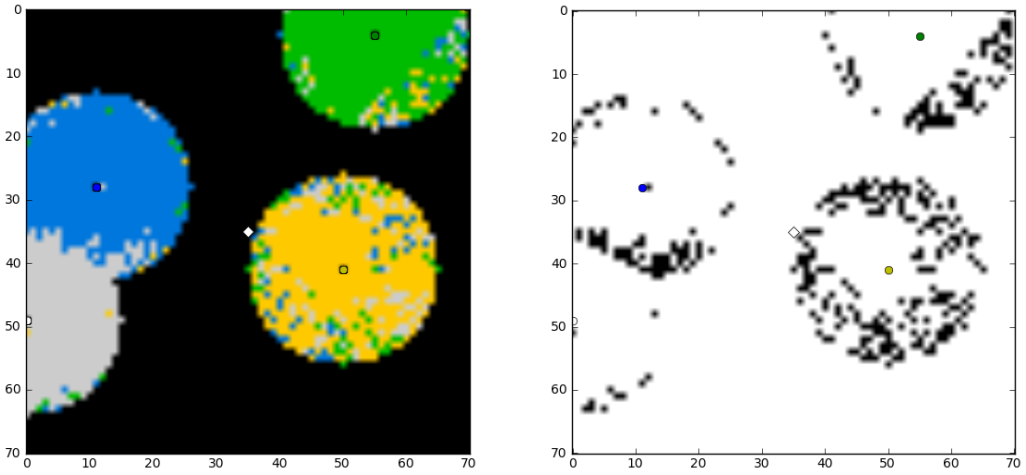
(b) After 1,259,000 simulations

(c) After 1,888,500 simulations

(d) After 2,518,000 simulations

Figure 9.16: A 71×71 grid after 500 simulations per state and firing distance 15

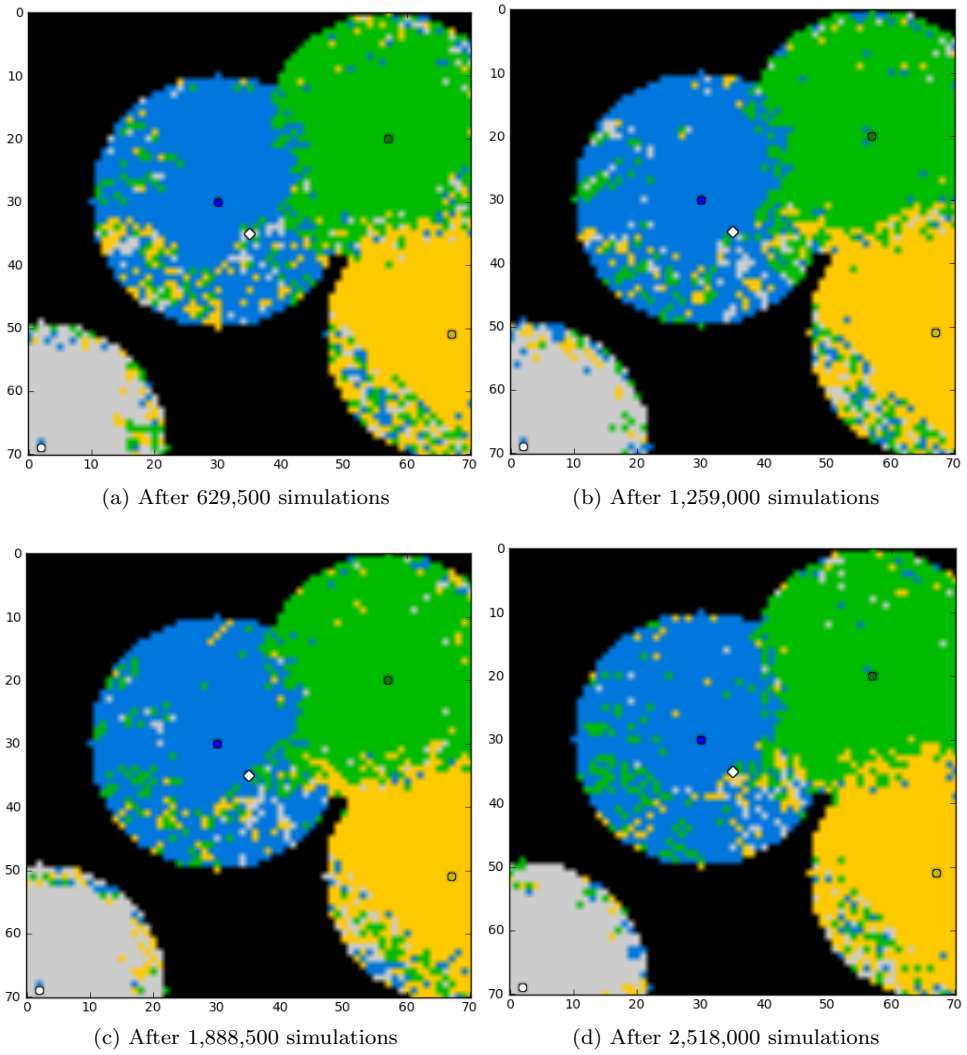
Figures (9.16)(a)-(d) also shows a slow learning rate when a firing distance of 15 is used. Figure (9.17) shows the errors made and the error percentage is 6.903%.



(a) Results after 2,518,000 simulations

(b) Errors after 2,518,000 simulations

Figure 9.17: A 71×71 grid with 500 simulations per state and firing distance 15



(a) After 629,500 simulations

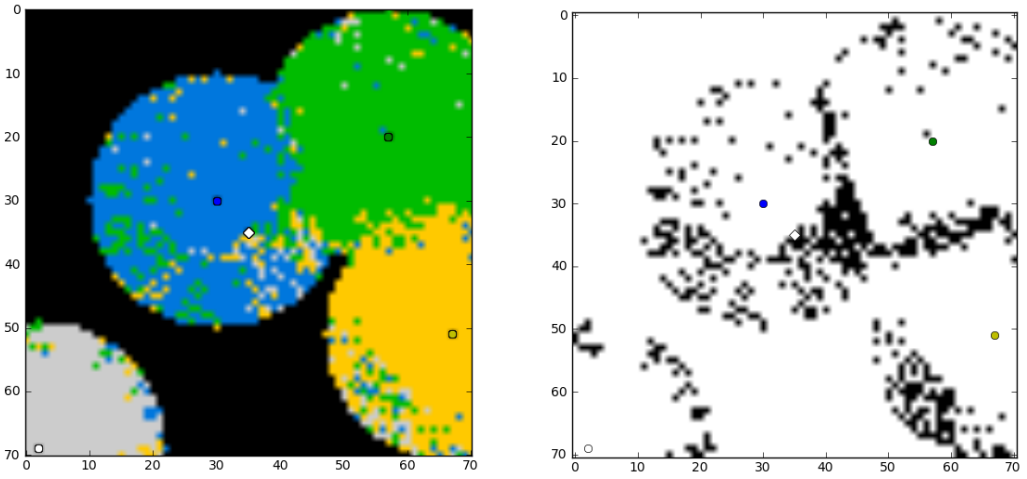
(b) After 1,259,000 simulations

(c) After 1,888,500 simulations

(d) After 2,518,000 simulations

Figure 9.18: A 71×71 grid after 500 simulations per state and firing distance 20

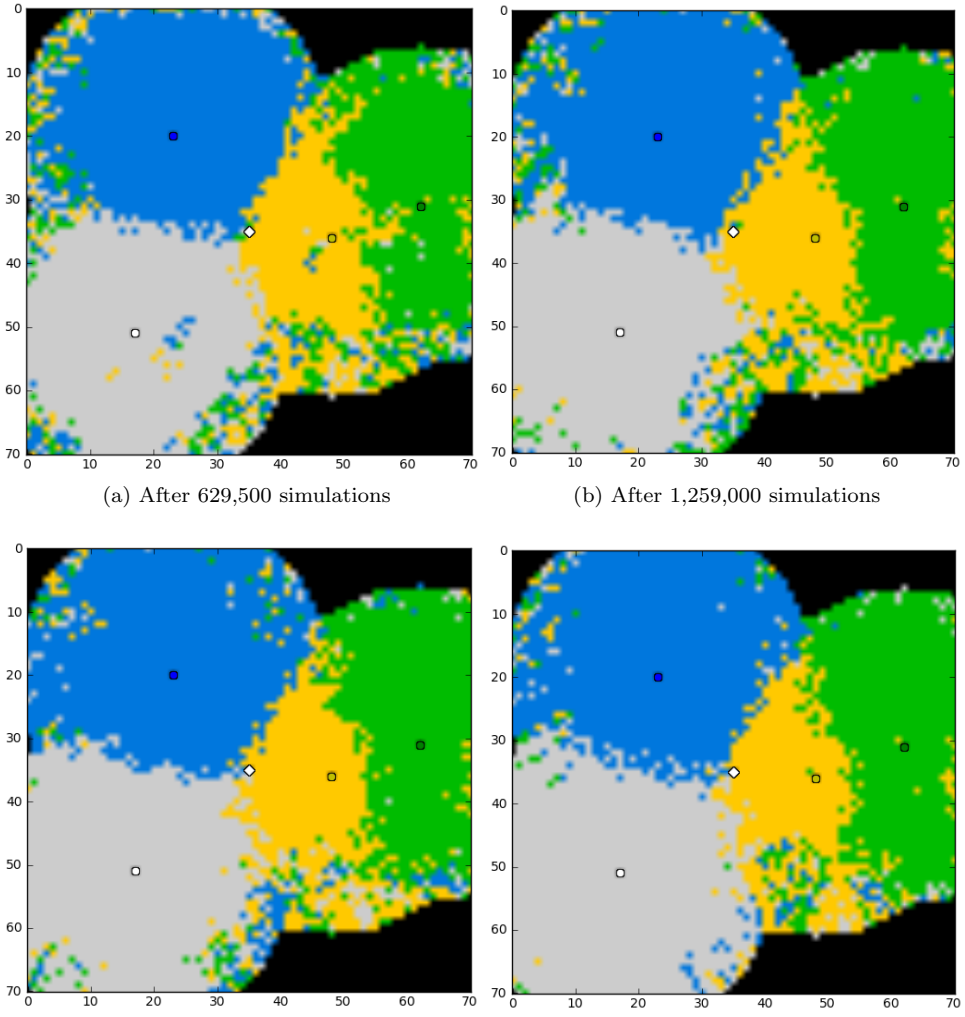
Figure (9.18)(a)-(d) also shows a slow learning rate, and the error percentage is 9.919% for a firing distance of 20. The errors made are shown in Figure (9.19).



(a) Results after 2,518,000 simulations

(b) Errors after 2,518,000 simulations

Figure 9.19: A 71×71 grid with 500 simulations per state and firing distance 20



(a) After 629,500 simulations

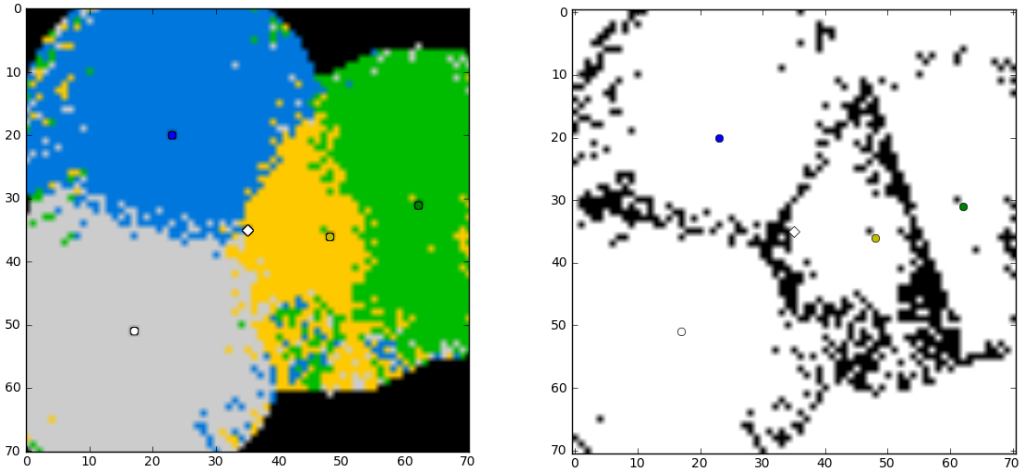
(b) After 1,259,000 simulations

(c) After 1,888,500 simulations

(d) After 2,518,000 simulations

Figure 9.20: A 71×71 grid after 500 simulations per state and firing distance 25

Figure (9.20)(a)-(d) also shows a slow learning rate, and the errors made amounts to 11.744% for a firing distance of 25. Figure (9.21) shows the errors made.

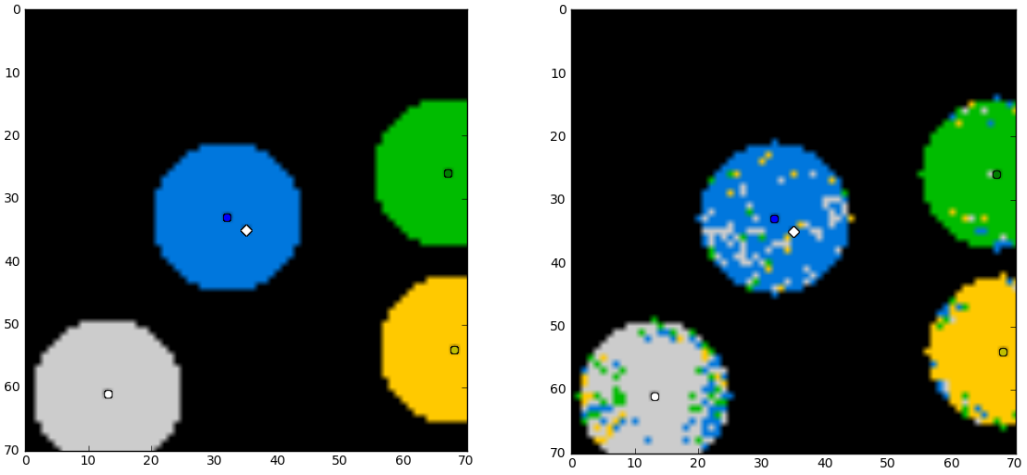


(a) Results after 2,518,000 simulations (b) Errors after 2,518,000 simulations

Figure 9.21: A 71×71 grid with 500 simulations per state and firing distance 25

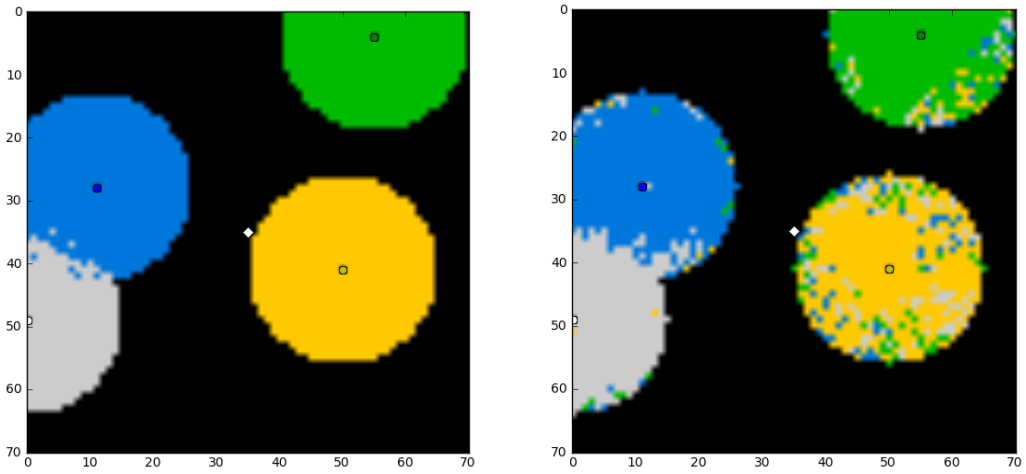
9.5 Discussion of results

Figures (9.22)-(9.25) show the side-by-side comparison for each of the completed simulations of the 71×71 grid examples.



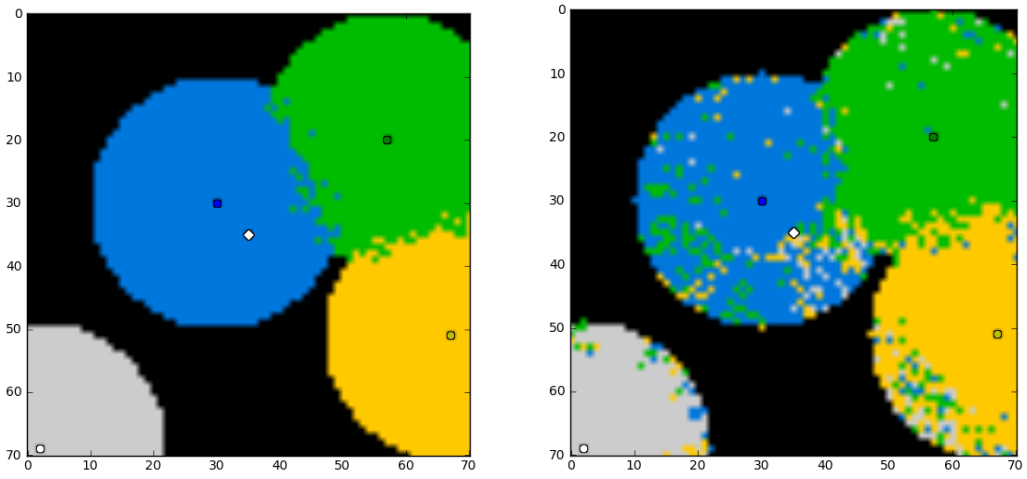
(a) MC: after $\approx 252,000$ simulations (b) Q-learning: after $\approx 2,518,000$ simulations

Figure 9.22: A 71×71 grid with firing distance 12



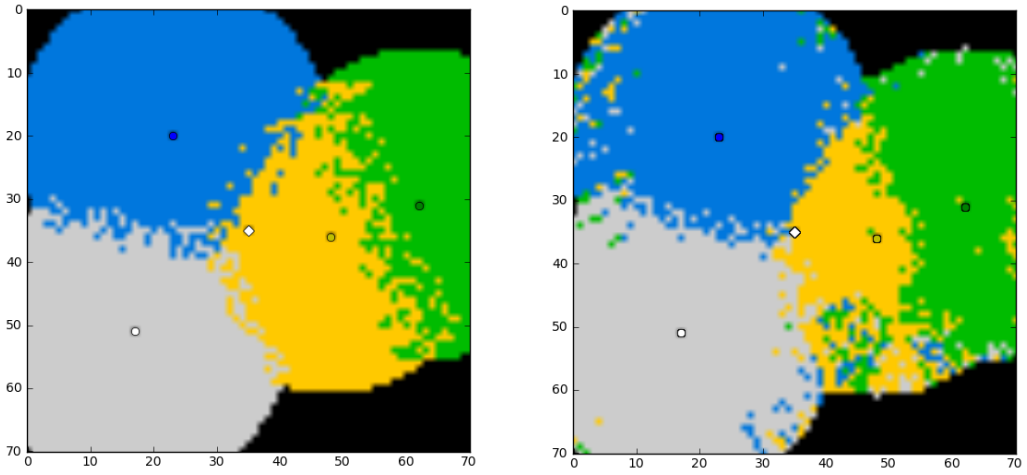
(a) MC: after $\approx 1,300,000$ simulations (b) Q-learning: after $\approx 2,518,000$ simulations

Figure 9.23: A 71×71 grid with firing distance 15



(a) MC: after $\approx 1,000,000$ simulations (b) Q-learning: after $\approx 2,518,000$ simulations

Figure 9.24: A 71×71 grid with firing distance 20



(a) MC: after $\approx 806,000$ simulations (b) Q-learning: after $\approx 2,518,000$ simulations

Figure 9.25: A 71×71 grid with firing distance 25

The results are fairly poor when applying the Q-learning algorithm to this problem. In Sutton and Barto (1998*a*) we read that the learned action-value function, Q, has been shown to converge to the optimal action-value function Q^* with probability one. This is clearly not what is happening here. The cause of this is that the best parameter-value combination has not been found yet.

9.6 Concluding remarks

In this chapter we applied both the MC control algorithm using exploring starts and the off-policy TD control algorithm, Q-learning, to the WA problem. Although the MC algorithm takes longer to execute, it gives better results than the Q-learning algorithm.

Table 9-II shows a comparison between the error percentages for both algorithms.

	Error % for various firing distances			
	12	15	20	25
MCES	0.258%	0.833%	1.508%	5.991%
TD Q	4.483%	6.903%	9.919%	11.744%

Table 9-II: Error percentages

Chapter 10

Conclusions and Future Work

10.1 Introduction

In this final chapter we present the reader with our main research findings as well as possible areas for future work.

10.2 Research findings

In this dissertation the machine learning field of Reinforcement Learning (RL) was studied. After building a coherent background, we wanted to know whether RL algorithms would yield promising results on the simplified version of the Weapon Assignment (WA) problem. We applied a Monte Carlo (MC) control algorithm with exploring starts (ES) to 7×7 and 71×71 grids, after which the off-policy Temporal-Difference (TD) learning control algorithm, Q-learning, was applied to 71×71 grids.

For the case where the grid size is 71, simulations were run with different firing distances, namely 12, 15, 20 and 25. It was found that the smaller the firing distance relative to the grid size, the longer the simulations took. This is because a small firing distance leads to much longer episodes. However, a smaller firing distance incurs a smaller error percentage. The bigger the firing distance, the greater the chance of overlapping between the weapons and the bigger the error percentage.

For the MCES algorithm a fixed discount parameter of $\gamma = 1$ was used. This algorithm performed exceptionally well and incurred only minor error percentages. When applied to a 71×71 grid with 200 simulations per state, the MCES algorithm took twice as long as the Q-learning algorithm with a firing distance of 15, and more than three times as long with a firing distance of 25.

Setting the discount parameter to $\gamma = 1$ caused the Q-values to diverge in the Q-learning algorithm. A greedy policy was followed while the learning rate (α) and the discount parameter (γ) was varied. It was determined experimentally that $0.1 \leq \alpha \leq 0.2$ yielded better results than larger α-values. These α-values were used with different γ-values and it was found that medium to large γ-values worked best. It was found that with a greedy policy, $\alpha = 0.1$ gave adequate results when applied along with $0.4 \leq \gamma \leq 0.7$.

Even though the MCES algorithm is considerably slower, it outperformed the Q-learning algorithm in all examples considered. The problem of simulation time can be rectified by code optimisation. We thus conclude that RL, especially the MCES algorithm, is a promising field to consider in the solving of the WA problem.

10.3 Future work

The TEWA problem was broken down into the two constituent elements of Threat Evaluation (TE) and Weapon Assignment (WA). This is arguably flawed, as the whole process must be considered at the same time, or at least in a closed loop of action. Currently this is not being attempted for obvious reasons of computational complexity. This underscores the general complexity of this process in Ground Based Air Defense Systems (GBADS), and indeed in many other Command and Control (C2) related problems, where the Observe-Orient-Decide-Act (OODA) loop is the primary process.

The problem of considering both TE and WA within the context of OODA is non-trivial and future work could focus on bringing these elements together and considering the computational burden that would result. The obvious question that comes to mind then is how one could find optimal policies that are developed in timely fashion to support the commander during an operation. For example, is it possible to rewrite the current algorithm for a massively parallel processing machine (such as a large set of Reduced Instruction Set Computer (RISC) machines) to achieve near-real time assignment and fire policy discovery?

Another suggestion for future work would include designing a more complicated grid by including different terrains and "obstacles" such as mountains and valleys, *et cetera*. Time delays for weapons could also be added, or even the constraint that certain weapons have limited ammunition. Another constraint could be that certain weapons can shoot further than others, where we currently have a constant firing distance across the board in a given problem. Another suggestion would be to consider the case where multiple weapons shoot at the same time.

In this dissertation we followed a greedy policy for the Q-learning algorithm. A possible improvement on this would be to follow an ϵ-greedy policy and vary ϵ along with γ and α. Combining the techniques investigated and tested in this work with other techniques in Artificial Intelligence (AI) and modern computational techniques may hold the key to solving some of the problems we now face in warfare.

Bibliography

Azak, M. and Bayrak, A. (2008). A New Approach for Threat Evaluation and Weapon Assignment Problem, Hybrid Learning with Multi-Agent Coordination. In: *23rd International Symposium on Computer and Information Sciences (ISCIS)*.

Bellman, R. (1957*a*). *Dynamic Programming*. Princeton University Press.

Bellman, R. (1957*b*). A markov decision process. *Journal of Mathematical Mechanics*, vol. 6, pp. 679–684.

Benaskeur, A.R., Rhéaume, F. and Paradis, S. (2007). Target engageability improvement through adaptive tracking. *Journal of advances in information fusion*, vol. 2, no. 2, pp. 99–111. ISSN 1557-6418.

Bertsekas, D. (1979 March). A distributed algorithm for the assignment problem. Lab. for Information and Decision Systems Report, MIT, Working Paper.

Çetin, E. and Esen, S. (2006). A weapon-target assignment approach to media allocation. *Applied Mathematics and Computation*, vol. 175, pp. 1266–1275. ISSN 0004-3702.

Champandard, A. (2002). Reinforcement Learning. Available from: `http://reinforcementlearning.ai-depot.com`, [Last seen on 24 October 2009].

Cline, B.E. (2004). Tuning Q-learning parameters with a genetic algorithm. Available from: `http://www.benjysbrain.com/ThePond/Tuning.pdf`, [Last seen on 22 March 2010].

Dai, P. and Goldsmith, J. (2009).

Deep, K. and Plant, M. (2005). Maximisation of expected target damage value. *Defense Science Journal*, vol. 55, no. 2, pp. 133–139.

Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms*. Ph.D. thesis, Politecnico di Milano, Italy.

Email (1999). Discussion of numerical instabilities in the Gambler's problem. Available from: `http://www.cs.ualberta.ca/~sutton/book/gamblers.html`, [Last seen on 30 October 2009].

Engelbrecht, A.P. (2007). *Computational Intelligence: An Introduction*, chap. 6. 2nd edn. John Wiley & Sons.

Gasser, M. (2009). Introduction to Reinforcement Learning. Available from: `http://www.cs.indiana.edu/~gasser/Salsa/rl.html`, [Last seen on 28 October 2009].

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional.

Grant, K. (1993). Optimal resource allocation using genetic algorithms. Naval Review, Naval Research Laboratory, Washington, DC.

Hoskins, J.C. and Himmelblau, D. (1992). Process control via artificial neural networks and reinforcement learning. *Computers & Chemical Engineering*, vol. 16, no. 4, pp. 241–251.

Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Hu, J., Sasakawa, T., Hirasawa, K. and Zheng, H. (2007). A hierarchical learning system incorporating with supervised, unsupervised and reinforcement learning. In: Liu, D. (ed.), *Advances in neural networks: ISNN 2007*, chap. 25. Springer Berlin Heidelberg.

Le Roux, W.H., Louis, A.L. and Nel, J.J. (2006). Using Bayesian Networks to Extend Weapon Assignment Subsystems. In: *Proceedings of the Seventeenth Annual Symposium of the Patter Recognition Association of South Africa (PRASA)*, pp. 213–218.

Lee, Z.-J. and Lee, C.-Y. (2005). A hybrid search algorithm with heuristics for resource allocation problem. *Information Sciences*, vol. 173, pp. 155–167.

Leenen, L. and Hakl, H. (2009 January). TEWA: Integrating threat evaluation with weapon assignment. Council for Scientific and Industrial Research (CSIR) DPSS-SM-EDERI-MSDS-032 Rev 1.

Leng, J., Fyfe, C. and Jain, L.C. (2009). Experimental analysis on Sarsa(λ) and Q(λ) with different eligibility traces strategies. *J. Intell. Fuzzy Syst.*, vol. 20, no. 1,2, pp. 73–82. ISSN 1064-1246.

Livio, M. (2003 January). *The Golden Ratio: The Story of Phi, the World's Most Astonishing Number*. Broadway. ISBN 0767908155.

Madni, A.M. and Andrecut, M. (2009). Efficient heuristic approaches to the weapon-target assignment problem. *Journal of Aerospace Computing, Information and Communication*, vol. 6.

Minsky, M.L. (1954). *Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*. Ph.D. thesis, Princeton University.

Minsky, M.L. (1961). Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, vol. 49, pp. 8–30.

Mohan, C. and Shanker, K.A. (1994). A controlled random search technique for global optimization using quadratic approximation. *Asia-Pacific Journal of Operational Research*, vol. 11, pp. 93–101.

Naidoo, S. (2008 May). Applying military techniques used in threat evaluation and weapon assignment to resource allocation for emergency response - a literature survey. Available from: `http://www.orssa.org.za/wiki/uploads/Johannesburg/Naidoo2008.pdf`, [Last seen on 17 June 2009]. Presented as seminar for ORSSA JHB.

Paradis, S., Benaskeur, A., Oxenham, M. and Cutler, P. (2005). Threat evaluation and weapons allocation in network-centric warfare. In: *2005 7th International Conference on Information Fusion (FUSION)*. IEEE, Piscataway, NJ, USA. ISBN 0 7803 9286 8. 2005 7th International Conference on Information Fusion (FUSION), 25-28 July 2005, Philadelphia, PA, USA.

Potgieter, J.J. (2007). *Real-time weapon assignment in a ground based air defence environment*. Master's thesis, Applied Mathematics, University of Stellenbosch, Stellenbosch, South Africa.

Powell, W.B. (2007). *Approximate dynamic programming: Solving the curses of dimensionality*, chap. 4.7.2. Wiley-Interscience.

Project TEWA (2007). TEWA article published in the Operations Research Society of South Africa (ORSSA) newsletter.

Puterman, M.L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Inc., New York. A Wiley-Interscience Publication.

Rosenberger, J.M., Hwang, H.S., Pallerla, R.P., Yucel, A., Wilson, R.L. and Brungardt, E.G. (2005). The generalized weapon target assignment problem. In: *10th International Command and Control Research and Technology Symposium*.

Roux, J. and Van Vuuren, J.H. (2007). Threat evaluation and weapon assignment decision support: A review of the state of the art. *ORiON*, vol. 23, no. 2, pp. 151–187. ISSN 0529-191-X.

Russell, S.J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. 2nd edn. Pearson Education.

Sikanen, T. (2008 November). Solving weapon target assignment problem with dynamic programming. Independent research project in applied mathematics (Helsinki University of Technology).
Available at: `http://www.sal.tkk.fi/Opinnot/Mat-2.108/pdf-files/esik08b.pdf`

Sutton, A. and Barto, R. (1998*a*). *Reinforcement Learning, an introduction*. 1st edn. The MIT Press.

Sutton, R. (2004). Reinforcement Learning FAQ: Frequently Asked Questions about Reinforcement Learning. Available from: `http://web.cs.ualberta.ca/~sutton/RL-FAQ.html`, [Last seen on 9 September 2009].

Sutton, R. and Barto, A. (1998*b*). Reinforcement Learning. Powerpoint slides Available from: `http://www.cs.ualberta.ca/~sutton/book/the-book.html`, [Last seen on 15 January 2009].

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, vol. 38, no. 3.

Watkins, C.J.C.H. and Dayan, P. (1992). Technical note Q-learning. *Machine Learning*, vol. 8, pp. 279–292.

Werbos, P. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 17, pp. 7–20.

Winston, W.L. (2004). *Operations Research: Applications and Algorithms*. 4th edn. Brooks/Cole, a division of Thomson Learning.

Woergoetter, F. and Porr, B. (2008). Reinforcement Learning. *Scholarpedia*, vol. 3, no. 3, p. 1448.

Zhang, W. (1995). A reinforcement learning approach to job-shop scheduling. In: *In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1114–1120. Morgan Kaufmann.

Appendix A

Dynamic Programming

A.1 The policy improvement theorem

Let π and π' be any two deterministic policies such that, for all states $s \in \mathcal{S}$,

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s). \tag{A.1}$$

Then the policy π' must be as good as, or better than, π. If this is true, it has to obtain a higher or equal expected return from all states $s \in \mathcal{S}$, thus:

$$V^{\pi'}(s) \geq V^{\pi}(s). \tag{A.2}$$

Starting from (A.1), we keep expanding the left hand side of the equation and obtain the result in (A.2):

$$\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\
&= E_{\pi'}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) | s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma E_{\pi'}\{r_{t+2} + \gamma V^{\pi}(s_{t+2})\} | s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) | s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) | s_t = s\} \\
&\vdots \\
&\leq E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots | s_t = s\} \\
&= V^{\pi'}(s).
\end{aligned} \tag{A.3}$$

If there is a strict inequality at (A.1) at any state, then there must be strict inequality

of (A.2) at one or more states. Suppose we have two policies: an original deterministic policy π and a changed policy π'. Policy π' is identical to π except that $\pi'(s) = a$ and $\pi(s) \neq a$ (a is in π' but not in π). Equation (A.1) holds at all states other than s. Thus if $Q^\pi(s, a) > V^\pi(s)$, the changed policy is indeed better than π (Sutton and Barto, 1998*a*).

Appendix B

Monte Carlo methods

B.1 On-policy Monte Carlo control

Let π' be an ϵ-greedy policy. From (A.1) and (A.2) it follows that $V^{\pi'}(s) \geq Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s)$. Using this relation, the conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned}
V^{\pi'}(s) &\geq Q^{\pi}(s, \pi'(s)) \\
&= \sum_a \pi'(s,a) Q^{\pi}(s,a) && \text{(B.1)} \\
&= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a Q^{\pi}(s,a) + (1-\epsilon) \max_a Q^{\pi}(s,a) && \text{(B.2)} \\
&\geq \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a Q^{\pi}(s,a) + (1-\epsilon) \sum_a \frac{\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} Q^{\pi}(s,a) && \text{(B.3)} \\
&= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a Q^{\pi}(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a Q^{\pi}(s,a) + \sum_a \pi(s,a) Q^{\pi}(s,a) && \text{(B.4)} \\
&= \sum_a \pi(s,a) Q^{\pi}(s,a) && \text{(B.5)} \\
&= V^{\pi}(s). && \text{(B.6)}
\end{aligned}$$

Equation (B.1) follows from our definition in equation (3.8). Equation (B.2) is the total actions taken; the non-greedy actions plus the greedy actions. In (B.3), instead of taking the maximum over all the actions, the sum is a weighted average with non-negative weights summing to one. This must be less than or equal to the largest number averaged. The weights are

$$\sum_a \frac{\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon},$$

where $\sum_a \pi(s,a) = 1$. Equation (B.7) shows that these weights indeed sums to one.

$$\begin{aligned}
\sum_a \frac{\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} &= \frac{1}{1-\epsilon}\left[\sum_a \pi(s,a) - \sum_a \frac{\epsilon}{|\mathcal{A}(s)|}\right] \\
&= \frac{1}{1-\epsilon}\left[1 - \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a 1\right] \\
&= \frac{1}{1-\epsilon}\left[1-\epsilon\right] \\
&= 1
\end{aligned} \tag{B.7}$$

By splitting the last term of (B.3) in two and simplifying it, we obtain (B.4). The first two terms of (B.4) are negatives of one another, thus cancelling out. We know from previous discussions that (B.5) is just the value function $V^\pi(s)$, proving our result.

Appendix C

Eligibility traces

C.1 The equivalence of forward and backward views

We show here that off-line TD(λ), as defined mechanistically, achieves the same weight updates as the off-line λ-return algorithm (Sutton and Barto, 1998*a*). We align the forward (theoretical) and backward (mechanistic) views of TD(λ).

Let $\Delta V_t^{\lambda}(s_t)$ denote the update at time t of $V(s_t)$. Let $\Delta V_t^{\mathrm{TD}}(s)$ denote the update at time t of state s according to the mechanistic definition of TD(λ) as given by (6.4). The goal is to show that the sum of all the updates over an episode is the same for the two algorithms:

$$\sum_{t=0}^{T-1} \Delta V_t^{TD}(s) = \sum_{t=0}^{T-1} \Delta V_t^{\lambda}(s_t) \mathcal{I}_{ss_t}, \; \forall \; s \in \mathcal{S}, \tag{C.1}$$

where $\mathcal{I}_{ss_t}$ is an identity-indicator function, equal to one if $s = s_t$, and equal to zero otherwise.

Note that an accumulating eligibility trace can be written explicitly as

$$e_t(s) = \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}_{ss_k}.$$

Thus, the left side of (C.1) can be written as

$$\begin{aligned} \sum_{t=0}^{T-1} \Delta V_t^{TD}(s) &= \sum_{t=0}^{T-1} \alpha\delta_t \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}_{ss_k} \\ &= \sum_{k=0}^{T-1} \alpha \sum_{t=0}^{k} (\gamma\lambda)^{k-t} \mathcal{I}_{ss_t} \delta_k \end{aligned}$$

$$\begin{aligned} &= \sum_{t=0}^{T-1} \alpha \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \mathcal{I}_{ss_t} \delta_k \\ &= \sum_{t=0}^{T-1} \alpha \, \mathcal{I}_{ss_t} \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \delta_k. \end{aligned} \tag{C.2}$$

Now we turn to the right-hand side of (C.1). Consider an individual update of the λ-return algorithm:

$$\begin{aligned} \frac{1}{\alpha}\Delta V_t^\lambda(s_t) &= R_t^\lambda - V_t(s_t) \\ &= -V_t(s_t) + (1-\lambda)\lambda^0[r_{t+1} + \gamma V_t(s_{t+1})] \\ &+ (1-\lambda)\lambda^1[r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})] \\ &+ (1-\lambda)\lambda^2[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V_t(s_{t+3})] \\ &\quad \vdots \qquad \vdots \qquad \vdots \qquad \ddots \end{aligned}$$

Consider all the r_{t+1}'s with their weighting factors of $1-\lambda$ times powers of λ in the first column inside the brackets. All the weighting factors sum to one. Thus we can pull out the first column and get an unweighted term of r_{t+1}. A similar trick pulls out the second column in brackets, starting from the second row, which sums to $\gamma\lambda r_{t+2}$. Repeating this for each column, we get

$$\begin{aligned} \frac{1}{\alpha}\Delta V_t^\lambda(s_t) &= -V_t(s_t) \\ &+ (\gamma\lambda)^0[r_{t+1} + \gamma V_t(s_{t+1}) - \gamma\lambda V_t(s_{t+1})] \\ &+ (\gamma\lambda)^1[r_{t+2} + \gamma V_t(s_{t+2}) - \gamma\lambda V_t(s_{t+2})] \\ &+ (\gamma\lambda)^2[r_{t+3} + \gamma V_t(s_{t+3}) - \gamma\lambda V_t(s_{t+3})] \\ &\vdots \\ &= (\gamma\lambda)^0[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] \\ &+ (\gamma\lambda)^1[r_{t+2} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})] \\ &+ (\gamma\lambda)^2[r_{t+3} + \gamma V_t(s_{t+3}) - V_t(s_{t+2})] \\ &\vdots \\ &\approx \sum_{k=t}^{\infty} (\gamma\lambda)^{k-t} \delta_k \\ &\approx \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \delta_k. \end{aligned}$$

The approximation above is exact in the case of off-line updating, in which case V_t is the same for all t. We have shown that in the off-line case the right-hand side of (C.1) can be

written as

$$\sum_{t=0}^{T-1} \Delta V_t^{\lambda}(s_t)\mathcal{I}_{ss_t} = \sum_{t=0}^{T-1} \alpha\mathcal{I}_{ss_t} \sum_{k=t}^{T-1} (\lambda\gamma)^{k-t}\delta_k,$$

which is the same as (C.2). This proves (C.1).

In the case of on-line updating, the approximation will be close as long as α is small. Thus V_t changes little within an episode. Even in the on-line case we can expect the updates of TD(λ) and of the λ-return algorithm to be similar.

C.2 n-Step TD prediction

Consider the backup applied to state s_t as a result of the state-reward sequence, $s_t, r_{t+1}, s_{t+1}, \ldots, r_T, s_T$. In Monte Carlo (MC) backups the estimate $V_t(s_t)$ of $V^{\pi}(s_t)$ is updated in the direction of the complete return:

$$R_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \ldots + \gamma^{T-t-1}\, r_T,$$

where T is the last time step of the episode and we call this quantity R_t, the target of the backup. In MC backups the target is the complete return, but in one-step backups the target is the first reward plus the discounted estimated value of the next state:

$$R_t^{(1)} = r_{t+1} + \gamma\, V_t(s_{t+1}),$$

where $\gamma\, V_t(s_{t+1})$ takes the place of the remaining terms $\gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \ldots + \gamma^{T-t-1}\, r_T$. This idea makes just as much sense after two steps as it does after one. The two-step target is $R_t^{(2)} = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, V_t(s_{t+2})$. Now $\gamma^2\, V_t(s_{t+2})$ takes the place of the terms $\gamma^2\, r_{t+3} + \gamma^3\, r_{t+4} + \ldots + \gamma^{T-t-1}\, r_T$. The n-step target is

$$R_t^{(n)} = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \ldots + \gamma^{n-1}\, r_{t+n} + \gamma^n\, V_t(s_{t+n}).$$

This quantity is sometimes called the *corrected n-step truncated return* which is a return truncated after n steps and then approximately corrected for the truncation by adding the estimated value of the nth next state. We instead refer to $R_t^{(n)}$ as the n-step return at time t.

If the episode ends in less than n steps, then the truncation in an n-step return occurs at the episode's end. This results in the conventional complete return. So if $T - t \leq n$, then $R_t^{(n)} = R_t^{(T-t)} = R_t$. The last n n-step returns of any episode are always complete returns, and an infinite-step return is always a complete return. This enables us to treat MC as the special case of infinite-step returns.

An n-step backup is defined to be a backup towards the n-step return. In the tabular state-value case, the increment to $V_t(s_t)$ (estimated value of $V^{\pi}(s_t)$ at time t), due to an n-step

backup of s_t, is defined as

$$\Delta V_t(s_t) = \alpha[R_t^{(n)} - V_t(s_t)],$$

where α is a positive stepsize parameter. Increments to the estimated values of the other states are $\Delta V_t(s) = 0 \forall\ s \neq s_t$. We define the n-step backup in terms of an increment, rather than as a direct rule, to distinguish two different ways of making the updates. In on-line updating, the updates are done during the episode, as soon as the increment is computed. In this case, $V_{t+1}(s) = V_t(s) + \Delta V_t(s)\ \forall\ s \in S$. In off-line updating, the increments are accumulated "on the side". They are not used to change value estimates until the end of the episode. In this case, $V_t(s)$ is constant within an episode, for all s. If its value in this episode is $V(s)$, then its new value in the next episode will be $V(s) + \sum_{t=0}^{T-1} \Delta V_t(s)$.

The expected value of all n-step returns is guaranteed to improve in a certain way over the current value function as an approximation to the true value function. For any V, the expected value of the n-step return using V is guaranteed to be a better estimate of V^π than V is, in a worst-state sense. The worst error under the new estimate is guaranteed to be less than or equal to γ^n times the worst error under V:

$$\max_s |E_\pi\{R_t^{(n)} | s_t = s\} - V^\pi(s)| \leq \gamma^n \max_s |V(s) - V^\pi(s)|.$$

This is called the *error reduction property* of n-step returns. Because of the error reduction property, one can show that on-line and off-line TD prediction methods using n-step backups converge to the correct predictions under appropriate technical conditions (Sutton and Barto, 1998*a*).